Needle : Leveraging Program Analysis to Analyze and Extract Accelerators from Whole Programs

Snehasish Kumar, Nick Sumner
Simon Fraser University,
{ska124, wsumner}@cs.sfu.ca

Vijayalakshmi Srinivasan
IBM Research
viji@us.ibm.com

Steve Margerm, Arrvindh Shriraman
Simon Fraser University,
{smargerm, ashriram}@cs.sfu.ca

***Abstract*—Technology constraints have increasingly led to the adoption of specialized coprocessors, i.e. hardware accelerators. The first challenge that computer architects encounter is identifying "what to specialize in the program". We demonstrate that this requires precise enumeration of program paths based on dynamic program behavior. We hypothesize that path-based [4] accelerator offloading leads to good coverage of dynamic instructions and improve energy efficiency. Unfortunately, hot paths across programs demonstrate diverse control flow behavior. Accelerators (typically based on dataflow execution), often lack an energy-efficient, complexity effective, and high performance (eg. branch prediction) support for control flow.**

We have developed *NEEDLE*, an LLVM based compiler framework that leverages dynamic profile information to identify, merge, and offload acceleratable paths from whole applications. *NEEDLE* derives insight into what code coverage (and consequently energy reduction) an accelerator can achieve. We also develop a novel program abstraction for offload calledBraid, that merges common code regions across different paths to improve coverage of the accelerator while trading off the increase in dataflow size. This enables coarse grained offloading, reducing interaction with the host CPU core. To prepare the Braids and paths for acceleration, *NEEDLE* generates software frames. Software frames enable energy efficient speculative execution on accelerators. They are accelerator microarchitecture independent support speculative execution including memory operations. *NEEDLE* is automated and has been used to analyze 225K paths across 29 workloads. It filtered and ranked 154K paths for acceleration across unmodified SPEC, PARSEC and PERFECT workload suites. We target *NEEDLE*'s offload regions toward a CGRA and demonstrate 34% performance and 20% energy improvement.

I. Introduction

While technology advances have enabled designers to assume a virtually unlimited number of transistors, power consumption per transistor no longer scales with feature size. As a result, *energy* and *power* compose the primary design constraints. Hardware acceleration, in the form of customized datapath and control circuitry for particular algorithms, may deliver the required performance and energy scaling [20]. Recently, major vendors [7], [12] have released multicore chips closely integrated with FPGAs. A central tenet of many accelerators [16], [18], [44] (either custom or reconfigurable) is their adoption of a dataflow-based execution model to elide instruction fetch and improve energy efficiency compared to the Out-of-Order (OOO) processor. However, such accelerators often rely on the OOO for either handling memory operations [17] and/or program control flow. This may lead to a reduction in energy efficiency for when offloading work to the accelerator at a fine granularity.

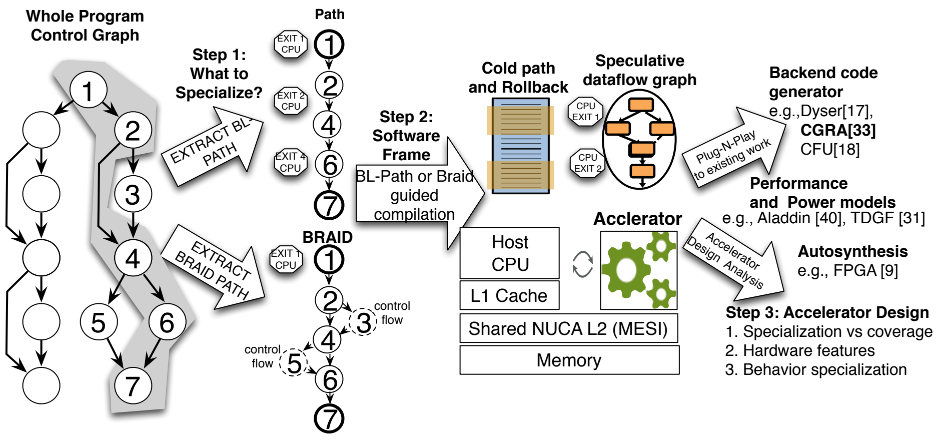
Often accelerators require an understanding of the specialized algorithm and program structure [37] to enable appropriate offload region formation. Since programs include complex control flow and have many possible execution paths, it is challenging to profile and compose an offload region for the accelerator. Real world examples of offloading a stable code region required that the API be redefined [36]. Recent works [31], [38], [40] have leveraged compiler intermediate representation (IR) to aid architectural simulation and enable comparison of different accelerator architectures. Such works still seem to largely leave unanswered the questions, "what code region in the original program should be specialized?" and "how to prepare it for offload?". Conventional profilers and analysis tools, e.g. gprof or trace analysis, are unsuitable for this task. Their scalability and accuracy is impeded by the structure of typical programs, which tend to have irregular control and dataflow. Compilers for coarse-grained reconfigurable array (CGRA) like fabrics [16], while successful for simple inner loops, find it challenging to prepare effective offload regions with many flows of control. VLIW compiler research has studied the formation of scheduling regions larger than a basic block by exploiting hardware predication (e.g., hyperblocks). Unfortunately, defining high quality regions depend on heuristics [3] and predication hardware; in Section II we analyze the specific requirements of accelerators.

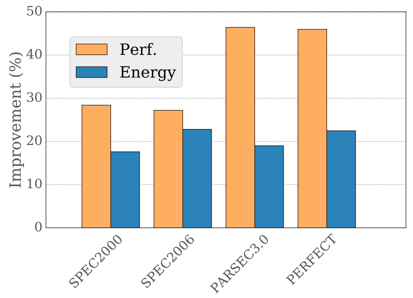
Our Insight and Proposal:

We demonstrate that an effective approach to building accelerators requires dynamic profiling for accurate early-stage exploration of the specialization tradeoffs between 1) targeting few code paths for efficiency and 2) coverage that seeks to offload a larger fraction of the application. We develop *NEEDLE*, an LLVM framework that leverages dynamic program analysis profiles to identify "what paths to specialize" in a program, merge paths and prepare them for acceleration. We study existing region formation algorithms (see Section II) and demonstrate the efficacy of Ball-Larus paths [4](BL-Path) for forming accelerator-friendly regions.

Figure 1 illustrates *NEEDLE*. **Step 1** analyzes programs to profile and construct two types of code regions for accelerators to target: BL-Paths and Braids. *BL-Paths* are single entry, single exit regions which represent a single flow of control. A control flow divergence leads to a jump back to the CPU and a reversion of externally visible program state.

Unfortunately, programs may execute a large number of





Paper Organization

Scope and Related work	§2
BL-Path extraction	§3
Braids: merging paths to reduce overhead	§4
Speculative execution on accelerators	§5
Performance and Energy evaluation	§6

Figure 1. Overview of the *NEEDLE* approach to define "what to Specialize". *NEEDLE* uses dynamic profiles to identify hot Ball-Larus [4] paths and merges them to create Braids improving code coverage. *NEEDLE* then generates software frames from the identified hot BL-Paths or Braids to enable speculative execution on accelerators. *NEEDLE* supports multiple backends; we evaluate a CGRA backend.

paths (over 100K in the workloads we study) with no single path dominating execution. This may lead to accelerators frequently switching between different paths, imposing a high overhead. To achieve high coverage we introduce a new program abstraction, called *Braid*, that takes advantage of the observation that many frequently executed BL-Paths tend to have the same basic blocks. *Braids* merge overlapping BL-Paths and seek to achieve high coverage. While BL-Paths revert to the CPU on any control flow divergence, the intuition behind Braids is that the program exits from the accelerator to the CPU only when the control flow appears to break out of a hot region of code. These regions are single-entry, single exit but incorporate multiple flows of control. BL-Paths and Braids are also inherently acyclic, we employ path prediction to identify loop back edges and construct larger regions for accelerator offload.

In **Step 2**, *NEEDLE* prepares the BL-Path and Braid abstractions to run on the hardware accelerator by generating *software frames* to handle control flow along the path and enable speculation on accelerators. This reduces the accelerator's reliance on the power-hungry OOO processor. Software frames support guarded execution on the accelerator [35]. *NEEDLE* creates frames by hoisting instructions in a BL-Path above the branches in that BL-Path, fusing them to create coarse-grained atomic regions of offload. The branches are converted into asynchronous guards that determine whether speculation was successful. *NEEDLE*'s frames permit all operations to be speculative, including memory operations. Software frames are accelerator microarchitecture independent and do not depend on specific hardware features (e.g., store buffers [18], [39]). *NEEDLE* regulates when the guards checks are inserted along the path to reduce the overheads of speculation failure while raising the number of hoisted operations to increase instruction parallelism.

We have analyzed over 225K paths across 29 workloads from three benchmark suites (PARSEC, SPEC, PERFECT) and analyzed the acceleration potential for 154K paths. *NEEDLE* automatically selects and generates the offload frames to target a coarse-grained reconfigurable fabric (CGRA). Overall, *NEEDLE* enabled offload improves performance by 34% and reduces overall energy by 20%. We contribute the following:

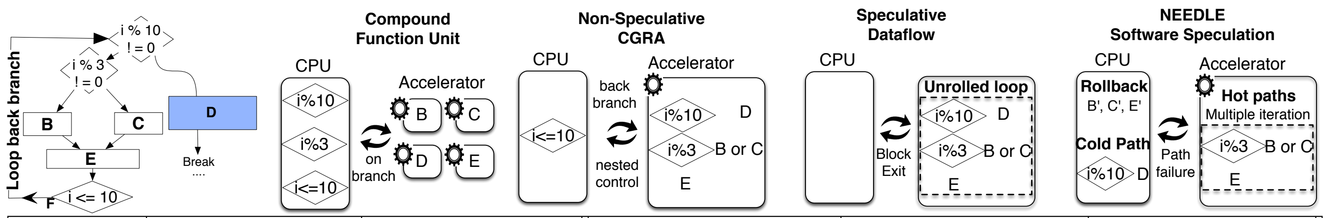
- *NEEDLE* is an automated tool chain that leverages dynamic workload profiles for the automatic selection and construction of "accelerator-friendly" regions. *NEEDLE* is target accelerator independent. We release the implementation as free and open source software [22].
- We introduce a new program abstraction, "Braids", that merges paths with common basic blocks to increase accelerator code coverage with less impact to hardware complexity and increased energy efficiency.
- *NEEDLE* generates software frames from hot paths and Braids to support guarded execution [35] on accelerators. This enables energy efficient software speculation and enlarges the granularity of offload to accelerators.

II. Scope and Related Work

NEEDLE is a profiling and compilation framework for sequential programs to target accelerators. A key impediment to implementing complexity effective hardware accelerators and precise code profiling is the control flow in sequential programs. Here, we study how *NEEDLE* can help existing accelerators handle multiple flows of control in a program with software controlled speculation. We also discuss the challenges with existing compiler abstractions for often used for accelerators, superblocks and hyperblocks.

A. Hardware Accelerator Perspective

Spatial accelerators often use a dataflow-based approach, custom or reconfigurable hardware, and use a compiler to map computation to functional units. Prior work has shown that code regions with regular control flow and abundant data parallelism achieve high performance and efficiency [8], [14], [17], [34]. However, sequential code with limited



		Compound Unit [10], [15], [18], [37]	Non Speculative CGRA [16], [32], [34], [41]	Speculative Dataflow [11], [42]	NEEDLE
Target Code	Granularity	Basic blocks ~~③~~	Hyperblocks	Hyperblocks	BL-Path [4] or Braid
	Scope	Few ops ~~③~~	Inner loops	Full Program	BL-Path or Braid
Design	Control flow	None	Predication ~~①~~	Dataflow predication ~~④~~	Software speculation
	Branch prediction	Necessary for high performance.			Not required
	Granularity	Fine	Medium (hyperblock)		Coarse (paths)
	Accelerator⇔OOO	High ~~②~~	Medium (on block termination) ~~②~~		Low (on spec. failure)
	Speculative Ops.	None ~~①~~	None ~~①~~	Partial (No mem ops)	Full
	Rollback Granularity	Small	Medium (entire hyperblock)		Flexible (sub-path)
Compiler		Static	Static or Profile-driven		Profile-driven
	Profiling	Superblock	Path-trees [16]	None	BL-Path or Braid
	Code-gen	Superblock	Hyperblocks	Hyperblocks	BL-Path or Braid
① Full speculation support. ② High energy efficiency. ③ Coarse-grained offload. ④ Low hardware overhead					

Figure 2. Comparison of sequential programs on spatial accelerator architectures

data parallelism, nested control-flow, and irregular memory access patterns either compromise on performance [8], [14], or energy efficiency [13]. Additionally, fine grained offload regions require frequent interaction with the OOO processor, leading to further energy waste. Figure 2 discusses the design trade-offs in spatial accelerators. Prior approaches can be broadly classified into three designs: i) compound function units with minimal or no support for control flow. ii) non-speculative CGRAs that leverage predication to handle forward branches, and iii) speculative dataflow adopted by block architectures that can execute backward and forward branches.

The compound function unit approach fuses frequently used operations but terminates the fusion at branches, limiting offload granularity to basic blocks. Larger granularity offloads can be constructed by either leveraging an OOO processor's branch predictor [24] or using apriori profiling techniques with superblock construction. As observed by prior work [30], such architectures (e.g., BERET [18]) when integrated with an out-of-order processor, require frequent interactions with the processor and achieve low ILP. The non-speculative dataflow approach is prevalent amongst CGRAs that include support for predicating individual operations. This design converts control flow into dataflow dependencies through if-conversion and hyperblock formation. Many challenges remain including support for speculating on backward branches, conversion of nested ifs, occupation of hardware resources, and lengthening of critical path [3]. Dataflow architectures such as TRIPS [42] target whole programs and support forward and backward branches at the expense of increased hardware complexity. The TRIPS compiler relies on aggressive loop unrolling and flattening to reduce backwards branches and removes forward branches by

Table I
CONTROL FLOW CHARACTERISTICS

Branch⇒Mem. Avg. mem ops dependent on a branch		
1—10	10 Apps	hmmer,lbm, crafty, bodytrack, mcf, fluidanimate, ferret, sar-back, gcc
>10	8 apps	gzip, blackscholes, h264ref, swaptions, vpr ,sar-pfa-interp1, povray, sjeng .
Mem⇒Branch. Avg. mem ops a branch is dependent on.		
1—10	11 apps	art, parser, lbm, bodytrack, bzip2, freqmine, gcc, h264ref, mcf, blackscholes, mcf
>10	7 apps	crafty, gzip, vpr, sar-pfa, povray, swaptions, sjeng
Max. predication. #Bits required for hot path if-conversion		
>100	13 apps	povray,fluidaimate, bodytrack, ferret, hmmer,sar-pf, art,crafty,fft-2d,sar-back, sjeng, swaptions, bzip2,vpr.
Loops. Number of backward branches in hot function.		
>10	14 apps	streamcluster, art, gcc, ferret, blackscholes, mcf2k, mcf2k6, hmmer, bodytrack, crafty, povray, swaptions, bzip2, vpr

forming hyperblocks. As a result of the increased hardware complexity, TRIPS exhibits only a 9% improvement in energy efficiency compared to an IBM Power4 superscalar processor at roughly the same performance [14].

In the remainder of this section we summarize the challenges posed by control flow in real world programs. See Section III for workload specific statistics. Table I summarizes the number of predication bits required to if-convert the fully inlined hottest function. Nine workloads required > 100 bits of predication. Only four workloads required < 10 bits. Our predication statistics differ from prior work [32] because of aggressive inlining of call sequences. Prior work would need inter-procedural analysis prior to if-conversion to reveal this behavior. We studied the Hyperblock sizes for all the inner loops in our function assuming two bits for predication [16]. We find that Hyperblocks only attain $\simeq$2.2$\times$ the basic block granularity. For four applications, sjeng, sar-pfa-interp1 and swaptions, hyperblocks increased block size by 6.3$\times$. Overall, predication and Hyperblocks do not suffice to enlarge the offload region granularity and minimize interactions with the OOO processor. To understand whether speculation is required, we look at individual branches and classify them into two categories (see Table I) `MEM-Branch` (`Ifs` that depend on memory operations) and `Branch-MEM` (`Ifs` statements with memory operations dependent on the branch). Either case could introduce serialization and loss of ILP. We find that on average each branch includes > 10 memory ops per branch in 8 applications (including floating point intensive applications). In 18 workloads, the MEM-Branch ratio is > 1 i.e., the branch depends on on at least one memory access. Both these statistics highlight the need for accelerators to implement a speculation framework including memory operations like an OOO processor. However, implementing hardware based speculation support is challenging in the absence of a notion of instruction or program order in dataflow accelerators. Thus we propose the use of *software based speculation*, where the compiler automatically inserts operations into the specialized region to support speculation (See Section V for details). Furthermore, workloads display varied characteristics and a unified hardware speculation strategy may be a poor fit. To summarize:

1) Control dependencies limit the granularity of offload to accelerators and hence require frequent switchbacks to an OOO processor, reducing the energy benefits.
2) Real programs have nested control flow, many backward branches and control interleaved with memory operations necessitating speculative execution support in accelerators.
3) Finally, some current accelerators rely on a OOO processors to leverage speculation which makes it challenging to enlarge the offload granularity.
4) Implementing speculation in accelerators is hindered by the need for a complex hardware mechanism to perform rollbacks which typically occur at fixed granularity.

Our Approach: *NEEDLE* adopts a software based speculation approach to compile specialized regions for accelerators. *NEEDLE* constructs atomic software frames from profiled hot regions. *NEEDLE*'s LLVM framework extracts each hot region into a separate "frame", converting biased branches along the path into guards [35]. *NEEDLE* generates the necessary rollback operations in software which enables workload tailored coarse-grained regions for offload. It improves energy efficiency and minimizes reliance on the OOO processor. *NEEDLE* supports full speculation, including memory operations.

B. Compilers for VLIW processors

NEEDLE's path-based offload region formation addresses a problem that at the high level seems similar to VLIW region formation strategies to handle control flow. Compilers for VLIW processors [1], [26], [27] pioneered the use of coarse-grained region formations by exploiting hardware predication. Hardware accelerators [16], [32] have primarily adopted "predication" to convert control flow dependencies into dataflow dependencies. Specialized regions offloaded to accelerators need to account for a large fraction of the dynamic instructions in order to achieve energy efficiency [19]. Our observations show, heuristic based region construction such as Superblocks [29] and Hyperblocks [27] targeted towards VLIW processors may include infrequently executed operations. It is unclear whether effective, coarse-grained offload regions can be constructed by tuning the heuristics in a manner independent of the accelerator and program control flow.

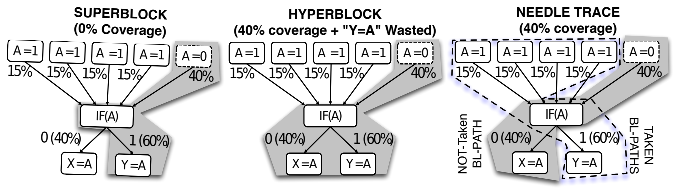


Figure 3. Superblock and Hyperblock construction for overlapped paths. % indicates the relative frequency.

Superblock and Hyperblock Construction Challenges: Some existing accelerators have sought to target offload regions [18], [21] that are constructed from edge profiles of branches. During offload region construction, a local decision is made at each branch for which side of the branch to include. In the presence of overlapping paths, edge profiles may yield less than optimal results. Figure 3 illustrates such an example. The edge profile will lead to a Superblock that will *always* fail and trigger a side exit. Hyperblock construction may recognize the lack of bias and fold in both sides. However, this will lead to wasted blocks (since given `A=0` the Y=A is wasted). Using Ball-Larus path profiling [4] provides a precise characterization of executed program paths. *NEEDLE* uses Ball-Larus paths (BL-Path) as a building block for offloaded regions. It is able to precisely identify the hottest path and construct the accelerator offload without waste.

Challenges in Achieving High Code Coverage. : We find 5 benchmarks out of 29, 403.gcc, 181.mcf, 429.mcf, and swaptions that demonstrate "infeasible" Superblock construction for innermost loops. In these cases, the constructed Superblocks do not correspond to the actual paths taken by the program. Overlapping paths misleadingly cause individual block edges to become hot even though that particular sequence of hot blocks may never appear in program execution. Infeasible Superblocks degrade performance and provide no acceleration coverage.

When Superblocks and Hyperblocks are feasible, they still may not capture the hottest paths through the program. The local branch edge profiles may skew the ranks of hot basic block sequences, deprioritizing the offloading of hotter program paths. Ranking the paths in order of frequency, we find 6 workloads (453.povray, 458.sjeng, 181.mcf, bodytrack,

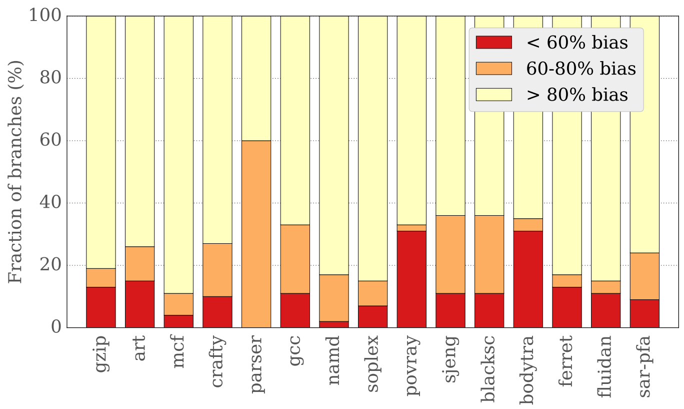


Figure 4. The distribution of biased branches in the application. Applications not shown in the plot have 99% of the branches with each branch > 80% bias.

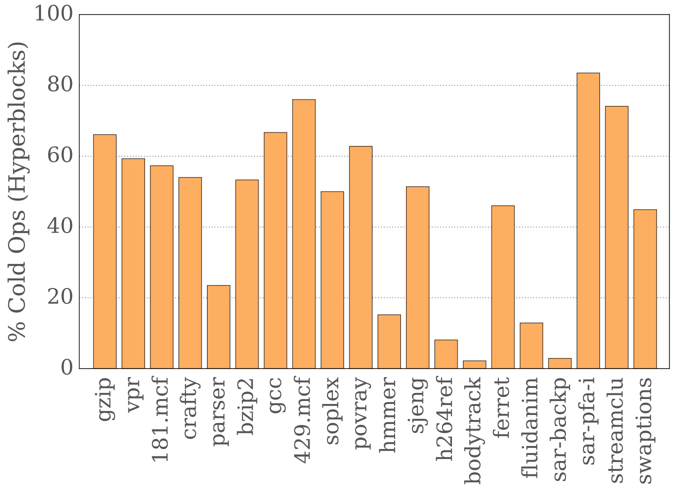


Figure 5. Fraction of "cold" ops included in Hyperblocks.

swaptions and 401.bzip2) where the constructed Superblocks are not the hottest path. This implies that there exists some path for the same program region that is executed more frequently than the Superblock.

Challenges in Heuristic Tuning: A key challenge for compilers seeking to leverage dynamic profiles is the tuning required for heuristic based approaches. To illustrate, we summarize the branch biases (i.e, how often a branch is taken) in the hottest function. The branch biases indicate to the compiler which successor basic block of a branch should be included (the taken or not-taken block). We find that in many workloads, 15 of 29, individual branch biases can vary significantly. Up to 24% of the branches have less than 80% bias (see Figure 4). In such cases, it is not clear how to tune the branch bias heuristic to achieve optimal coverage. Note that Superblock and Hyperblock formation requires carefully tuned heuristics [2], multiple metrics including resource utilization and execution coverage need to be considered. More important, the heuristic must understand how the included blocks will interact with other included blocks and the runtime behavior of shared branches in the offloaded region. Figure 5 plots the number of operations that are part of the Hyperblock but are "cold", i.e infrequently executed. For a hardware accelerator, such operations tend to waste both 1) resources leading to area penalty for custom circuit and/or 2) energy in reconfigurable accelerators. The Hyperblock construction makes a local decision and thus may include wasted operations (See Figure 3 for an example). Without contextual program path information they may include blocks without including the other basic blocks in the path.

Dynamic Compilation for Accelerators: Recent work has studied dynamic compilation to target a CGRA [46]. They however do not support control flow and map a single basic block to the CGRA at a time [46] . Control flow speculation is key to enabling coarse-grained regions that improve the effectiveness of a dynamic compiler. *NEEDLE* focuses on identifying such coarse-grained regions in programs.

III. BL-Path Accelerators

In this section we contrast the notion of a Ball-Larus path (BL-Path) and contrast it against other region formation strategies (e.g., Superblock or Hyperblock). We identify the BL-Path characteristics that make them suited for accelerators. In particular, the BL-Path approach will not encounter the same challenge as Superblocks in Figure 3 since it identifies not just the bias of the individual branch but the overall bias of the code path to reach the branch. This leads to accurate profiling of basic block hotness, formation of regions with guaranteed coverage of dynamic execution by *construction*, thus improving efficiency.

Ball-Larus path profiling [4] is used by *NEEDLE* to obtain the initial set of acyclic candidate paths that summarize a program's dynamic behavior. The Ball-Larus method pre-processes the control flow graph of a routine to replace loop back edges with fake edges, one each from entry to back edge target and from back edge source to routine exit. Paths in the directed acyclic graph are enumerated bottom-up using dynamic programming, leading to unique ids for each acyclic path. Instrumentation is inserted to track which paths are executed at runtime. The dynamic profile of executed paths is collected, i.e for each unique path id we log the number of times it has been executed. Each unique id can be decoded to a sequence of basic blocks. We rank each uniquely executed path and select most suitable candidates for acceleration. *NEEDLE* extracts the blocks in the selected path(s) into an offload function, adds support for software speculation and prepares it for hardware accelerator synthesis.

A. Path Ranking

To rank the paths for hardware accelerators, we define a new metric, path weight (P_{wt}). It captures both the execution frequency of the path and number of operations. Eliding instruction fetch is a primary source of energy efficiency in hardware accelerators [19]. Maximizing P_{wt} maximizes energy saved in the processor front-end. For the first order ranking of paths our weight metric assumes that all instructions carry the same weight, since instructions carry similar front-end energy costs in a processor. Latency of each instruction can be factored into the weight should the primary target be performance rather than energy efficiency. We also calculate Function Weight (F_{wt}), which accumulates all its constituent $P_{wt}s$. We present only the data for highest ranked function by weight for the sake of readability.

To understand the potential implications of selecting a frequency based metric, we profiled the time spent in the hottest ranked path using Linux's `pprof` (1500 samples/s) versus its parent function. We computed P_{wt}/F_{wt} as well as $P_{samples}/F_{samples}$ and compared the values as relative weights. In 12 of the 29 workloads we study, the sampling based profile indicated an average 10% increased weight; in 6 workloads we found a 15% decrease and in 4 workloads no change. The variability in sampling based profiling reaffirms our decision to use a frequency based metric that accounts for the dynamic power of the front-end.

B. BL-Path Properties

In the remainder of this section we present the characteristics of BL-Paths profiled across 29 workloads. Figure 6 shows the breakdown of dynamic instructions attributed to the top five paths amongst all paths in the highest ranked function. Table II presents the characteristics of the top five highest ranked BL-Paths.

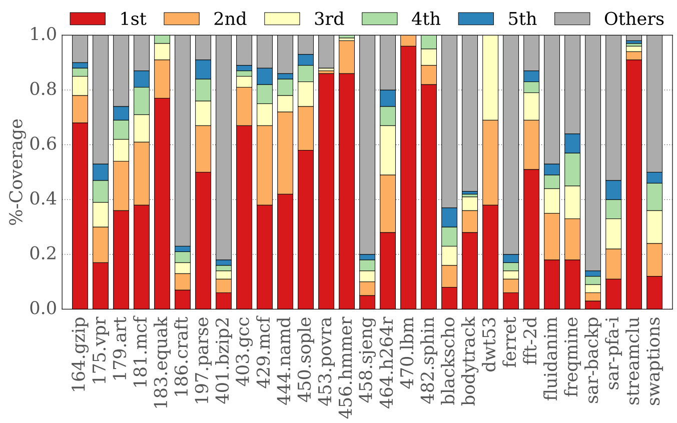


Figure 6. Path Coverage : Path weight (P_{wt}) by rank.

Few BL-Paths Enable High Coverage: Figure 6 shows the coverage (P_{wt}) of paths in our workloads. The stacks (bottom to top) represent the coverage of the highest to lower ranked paths. The average coverage (fraction of dynamic instructions) of the highest ranked BL-Path is 25%. In 18 of 29 applications the top path offers 20% or more coverage (See Figure 6). The median coverage using top five paths is 86%. Thus reasoning about paths allows us to understand the semantically different, yet frequent basic block sequences executed by a workload. For instance, as shown in Figure 3, it is desirable to precisely account for the frequency of the taken and not-taken sides based on which path is invoking the if-block.

BL-Paths Enable Coarse-Grained Offload (Table II:C3): Table II:C3 shows the average size (number of instructions) of the top five paths in the workloads. With a coarse-grained offload region, more computation is performed on the accelerator and fewer interactions with the host OOO processor. Note that BL-Paths are acyclic; we investigate techniques to enlarge them further in Section IV-B. The median size across workloads is 65 operations. We have highlighted the applications that had a large number of branches in the path despite which the BL-Path was able to construct regions with 80+ operations (outliers are swaptions – 438 and 458.sjeng – 50). The highlighted values in C4 indicate workloads in which the BL-Path traverses many branches. On 11 of the 29 workloads the highest ranked BL-Path spans across $\simeq$13 branches. Note that these sizes are larger than those observed with edge-profiled Superblocks in prior work [18]. An interesting workload is 401.bzip2, where the number of instructions in the top five paths vary significantly (29, 66, 371, 371, 194) with each path providing small coverage of the overall ($\sum_5 Cov. = 18$). Table 1:C7 indicates the number of memory operations that are part of the BL-Path and would be hoisted and become control independent when software frames are formed. The circled numbers highlight the workloads that benefit most from memory speculation.

BL-Paths Have Overlapping Basic Blocks (Table II:C8): A key concern with accelerator architectures is reusability or recurrence of acceleratable sections in the program. Prior work has evaluated this at the granularity of subgraphs containing a few operations [10], [45]. Here we present a methodical evaluation at the path granularity. Programs often execute a large number of paths in the same region. This often implies that many paths share common basic blocks. For instance, at individual branch merge points (e.g., In Figure 1 E occurs along both the ABEF and ACEF paths and F occurs along ABEF,ACEF and ACF). We quantify the overlap of basic blocks across the top five paths in each workload. Column C7 in Table II represents the average (geomean) block overlap. In 10 out of the 29 workloads we see that between 6–31 BL-Paths overlap (outlier: swaptions). In the other 19 workloads at least 2 paths overlap. BL-Paths enable precise accounting for common basic blocks.

Hardware overheads for BL-Path based accelerators (Table II:C5 & C6): The number of live inputs and outputs determine the amount of data transferred to and from the accelerator. We summarize the results for the top five paths in Table II:C5. These do not include the memory operations within the accelerator. Some workloads may have compute intensive regions with few live in and live out values (e.g., 470.lbm, 175.vpr, 183.equake, 444.namd). The workloads

Table II
PATH CHARACTERISTICS

C1 : Exe. Paths **C2** : $\sum_5 Cov.$: Coverage of top 5paths **C3** : Ins.
C4 : Branch **C5** : Live Vals **C6** : Phi ops cancel **C7** : Mem.ops
C8 : # Overlapping paths

		C1	C2	C3	C4	C5	C6	C7	C8
	Name	Exec	$\sum_5 Cov.$	#Ins.	$\Diamond$	$\downarrow,\uparrow$	$\not\phi$	Mem	Ov.
SPECINT and SPECFP	164.gzip	80	90	33	4	7 , 5	4	4	$\textcircled{6}$
	175.vpr	713	53	**80**	**8**	6 , 3	8	$\textcircled{21}$	2
	179.art	1446	74	24	2	3 , 4	2	7	$\textcircled{12}$
	181.mcf	48	87	30	2	5 , 3	2	7	2
	183.equake	7	100	**88**	1	9 , 5	1	$\textcircled{32}$	1
	186.crafty	37K	23	49	**7**	8 , 3	7	4	$\textcircled{31}$
	197.parser	10	91	33	3	6 , 2	3	6	2
	401.bzip2	54K	18	**207**	**15**	10, 6	15	29	$\textcircled{15}$
	403.gcc	21	89	43	4	7 , 5	4	6	3
	429.mcf	41	88	21	2	4 , 2	2	6	2
	444.namd	57	86	**90**	2	18, 10	2	$\textcircled{14}$	2
	450.soplex	67	93	33	2	7 , 3	2	7	3
	453.povray	375	88	**137**	**8**	7 , 4	8	17	$\textcircled{21}$
	456.hmmer	61	100	**105**	**6**	12, 2	6	$\textcircled{35}$	2
	458.sjeng	45K	20	**50**	**9**	3 , 3	9	8	$\textcircled{43}$
	464.h264ref	43	80	49	4	11, 3	4	9	2
	470.lbm	2	100	**232**	2	3 , 2	2	$\textcircled{45}$	2
	482.sphinx3	6	100	30	1	9 , 4	1	6	1
PARSEC and PERFECT	blackscholes	42	37	**380**	**19**	9 , 1	19	0	$\textcircled{11}$
	bodytrack	732	43	68	4	10, 5	4	3	$\textcircled{24}$
	dwt53	12	100	28	1	6 , 2	1	6	1
	ferret	556	20	**98**	**9**	7 , 6	9	2	$\textcircled{10}$
	fft-2d	29	87	38	2	6 , 3	2	4	2
	fluidanimate	377	53	67	4	9 , 4	4	10	5
	freqmine	22	64	31	2	6 , 4	2	10	2
	sar-back.	539	14	**85**	**9**	7 , 5	9	6	3
	sar-pfa-interp1	53	47	**146**	**14**	14, 3	14	8	$\textcircled{8}$
	streamcluster	42	98	35	3	6 , 4	3	6	2
	swaptions	11K	50	**438**	**29**	9 , 3	29	$\textcircled{32}$	$\textcircled{138}$

with coarse-granularity offload (C3 highlighted) have an average $\simeq$10 live ins and $\simeq$4 live outs). ϕ instructions in LLVM correspond to selection operator and incur significant hardware overhead [5]. When speculating on the control flow in a BL-Path (see Section V), a frame is constructed, ϕs can be removed. It is interesting that in 10 out of 29 workloads, we remove multiple ϕs per branch. This implies speculation on just a few branches we can significantly reduce hardware.

IV. BL-Path Expansion and Braids

In order to reduce execution migration between the host and the accelerator, we explore two approaches to increase the granularity of offload. *BL-Path Expansion* seeks to extend acceleration across back edges of loops, while *Braids* combine multiple paths for offloading.

A. BL-Path Target Expansion

BL-Paths are acyclic in nature. Sequencing paths across backward branches is essential to loop pipelining and extraction of data parallelism. Prior work [8], [28] has shown that outer loop pipelining is critical to finding parallelism in sequential programs. Acyclic regions superblocks and hyperblocks encounter the same challenge and typically attempt to grow in size via loop unrolling, branch target expansion and loop peeling [29]. *NEEDLE* can construct offload regions by sequencing multiple BL-Paths using the dynamic execution profiles. We collected a path trace (sequence of path ids) during the profiling phase of the program. We then processed the trace and found that in many cases applications demonstrate a bias for back-to-back paths. We use the profile to guide which path to sequence next.

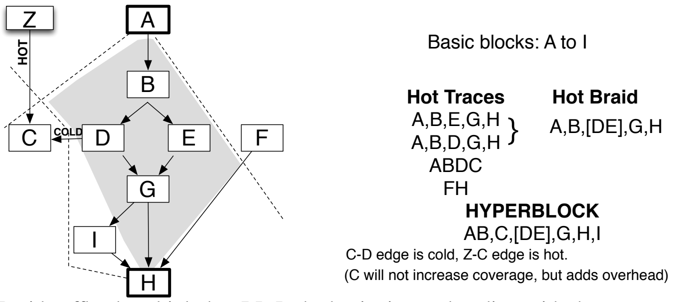
Table III
NEXT PATH TARGET EXPANSION

Path Seq. Bias	+Ops	Workloads
90-100%	68%	175.vpr 179.art 181.mcf 401.bzip2 403.gcc 429.mcf 444.namd 453.povray 456.hmmer 470.lbm 482.sphinx3 blackscholes dwt53 fft-2d streamcluster
70-90%	2×	183.equake 450.soplex 464.h264ref
<70%	73%	164.gzip 186.crafty 197.parser 458.sjeng bodytrack ferret fluidanimate freqmine sar-backprojection sar-pfa-interp1 swaptions

We summarize the data in Table III. In 15 out of 29 workloads, a single path occurred in sequence more than 90% of the time. Of these 10 workloads repeated the same BL-Path. This enables us to enlarge the granularity of offload by a factor of 2×. The remaining 5 workloads (*.mcf, 401.bzip2, 403.gcc, blackscholes) where a different path follows in sequence we were able to expand the offload by a further 17%. Overall, the same path repeats in 17 out of 29 workloads, and the average offload unit can be increased in size by 72%.

B. Braids – Merging BL-Paths

Each BL-Path offload targets a specific sequence of basic blocks (which corresponds to a program path) with a single flow of control; any deviation requires accelerator rollback. Programs may have many paths which originate from the same basic block. It is challenging to determine exactly which path should be invoked. The penalty for invoking the wrong accelerator is rollback. A promising approach would be to merge paths to create a single offload unit. The key questions are "which paths to merge?" and "how to merge paths?". We present a new offload region abstraction, *Braids*, formed by merging BL-Paths thus achieving coverage equal to the cumulative coverage of the BL-Paths. We analyzed all the paths across our workloads and observed that in many cases of overlap, in particular the paths had a common start and end basic block. These paths diverged from the same point in the program and then re-converged. Consider Figure 7, the BL-Paths for this section are `ABDGH` and `ABEGH` (all hot paths start at block A and exit at block H). We construct a Braid by merging BL-Paths, and this requires the introduction of multiple flows of control within the region. Note that the Braids are acyclic and thus introduce only forward branches. Braids include the basic blocks observed to have been executed and guarantees monotonic increase in coverage with each merged BL-Path. Braids are prevalent in program loops that have multiple control flows within the loop body. Since Braids only merge BL-Paths that share the entry and exit block, live ins and live out values do not change. This permits the accelerator to transparently switch between the BL-Path or Braid configurations based on code coverage and area tradeoffs.



Braids offload multiple hot BL-Paths beginning and ending with the same basic blocks. In contrast, Hyperblocks also fold in cold blocks C and I.

Figure 7. Braid construction from BL-Paths

Relationship to Hyperblocks [27], Path-Trees [6] : Hyperblocks are an extension to Superblocks where basic block successors to unbiased branches are merged for architectures that support predicated execution. Braids are a specific type of Hyperblocks that support multiple flows of control but always exit from the same block on completion. The heuristic based construction of Hyperblocks gives rise to multiple exits, which makes it challenging to bound the construction process. For example in Figure 7 the Hyperblock could include `C`. Additionally, Hyperblocks needs "tail duplication" for block `F` since it may merge paths that don't exit at F. Path trees are used by DySER [17]. In essence, they are Hyperblocks constructed from path profiles rather than edge profiles. They merge paths which originate from the same basic block and diverge. In Braids, the biased branches are converted to guards and enable speculation which is more energy efficient than predication when successful. While path trees originate from the same block, they may diverge to different basic blocks and have different live out sets based on the exiting blocks.

Braids improve accelerator code coverage (see Table IV:C3): We calculate the coverage-per-op ($\frac{C3}{C4}$) i.e., the fraction of dynamic execution covered by each operation in the Braid. This permits us to evaluate coverage by neutralizing the effect of a larger region size. Constructing Braids improved coverage-per-op for 17 applications (avg 0.85% of total dynamic execution per op). For 6 workloads the Braids improves coverage but also substantially increases the region size. For 444.namd, swaptions, 175.vpr, 470.lbm, 401.bzip, 186.crafty the BL-Path provided better coverage per op; Braid provided better coverage overall.

It might be beneficial to look beyond the hottest path and merge the lower-ranked paths to create hot Braids. This occurs in cases (eg.175.vpr, fluidanimate and sar-backprojection) where there is not much overlap between the hotter BL-Paths but there is a lot of overlap in the lower-ranked BL-Paths making them amenable for merging. *NEEDLE* provides a methodical framework to reason about this tradeoff.

Table IV
BRAID CHARACTERISTICS

C1 : Number of Braids **C2** : # paths merger to create a Braid
C3 : Code coverage **C4** : Ins. **C5** : Guards i.e., branches removed
C6 : IFs; branches introduced when merging paths **C7** : Live Vals

		C1	C2	C3	C4	C5	C6	C7
		#Braids	$\frac{\#Paths}{Braid}$	Cov%	#Ins.	◊	IFs	↓,↑
SPECINT and SPECFP	164.gzip	48	1.5	**80**	39	3	3	8 , 5
	175.vpr	549	1.2	28	177	12	10	8 , 2
	179.art	84	2.3	36	21	1	0	2 , 1
	181.mcf	40	1.1	38	53	3	3	6 , 2
	183.equake	8	1.0	**77**	144	1	0	14 , 8
	186.crafty	388	2.0	6	28	5	0	6 , 3
	197.parser	7	1.4	**49**	56	1	0	5 , 2
	401.bzip2	3383	1.4	5	27	4	0	7 , 3
	403.gcc	9	1.8	**73**	50	1	6	6 , 7
	429.mcf	39	1.0	37	31	3	1	6 , 2
	444.namd	51	1.1	42	229	1	0	36 , 16
	450.soplex	47	1.3	**57**	30	2	0	5 , 3
	453.povray	8	11.8	**85**	54	1	1	2 , 1
	456.hmmer	47	1.1	**85**	61	2	0	16 , 1
	458.sjeng	296	1.7	27	2272	36	115	3 , 3
	464.h264ref	40	1.1	33	71	6	1	16 , 5
	470.lbm	2	1.4	**100**	511	1	1	3 , 1
	482.sphinx3	7	1.0	**82**	30	1	0	9 , 3
PARSEC and PERFECT	blackscholes	4	5.3	**52**	381	16	8	9 , 1
	bodytrack	19	6.0	27	45	4	0	12 , 2
	dwt53	13	1.0	37	23	1	0	9 , 1
	ferret	95	1.6	39	138	7	5	6 , 6
	fft-2d	23	1.2	**51**	39	1	0	8 , 1
	fluidanimate	74	1.3	25	117	8	8	4 , 4
	freqmine	21	1.1	17	43	4	0	6 , 2
	sar-backprojection	125	1.3	19	135	4	8	6 , 6
	sar-pfa-interp1	9	2.0	**88**	344	14	14	14 , 3
	streamcluster	31	1.2	**91**	47	4	0	5 , 2
	swaptions	85	3.0	38	1704	82	42	9 , 3

Fewer guards than BL-Path ⇒ Fewer speculation failures: When merging paths, Braids introduce multiple flows of control within the region. This effectively reduces the number of guards that would have otherwise been needed by the constituent paths individually. On 12 applications the Braids have 2× fewer guards than the hot BL-Path. The reduction in guards directly correlates with how much overlap is between the merged BL-Paths. Fewer guards mean that the offloaded region into the accelerator is less likely to fail (see Section VI for the details). Outliers, 458.sjeng and swaptions increased the number of guards by 10× due to merging paths with minimal overlap, but each with path having many guards.

Braids enable memory speculation: The control dependency enforced by the branch may limit memory level parallelism available in the dataflow graph; eliminating branches from the hot region will enable memory operations to speculative execute. We find that in 14 workloads the hottest braids have 0 memory operations dependent on a branch (in comparison to 11; see Branch-Mem in Table Ib). The number of workloads where the dependencies were greater than 10 was reduced to 4 workloads. In 6 workloads (179.art, 186.crafty, 197.parser, 401.bzip2, bodytrack, freqmine) the number of dependencies reduced to zero in the hottest Braid.

V. Execution Model

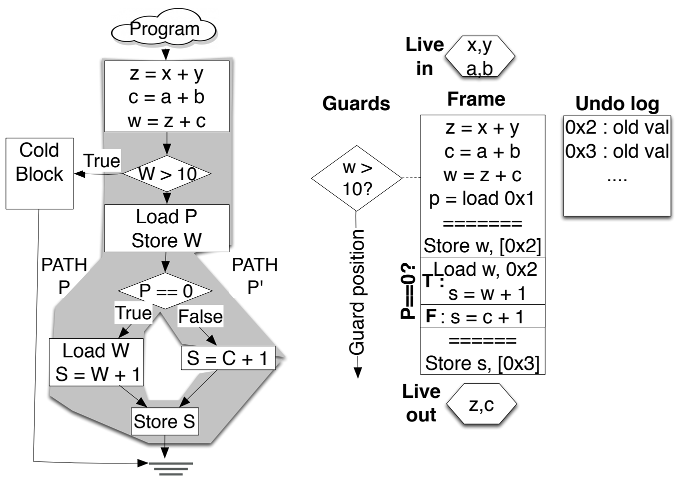


Figure 8. Frame construction from Braid.

In *NEEDLE*, candidate hot BL-Paths and Braids are converted into software frames for offload to the accelerator. They serve the same purpose as Traces within a Trace Scheduling compiler [25] or Superblocks [29] and hardware frames [39]. While Traces are multiple-entry, multiple-exit regions, Superblocks and hardware frames [39] are single-entry multi-exit regions with a single flow of control. *NEEDLE* constructs single-entry single-exit regions but also support multiple flows of control. Software frames are atomic and coarse-grained, enabling effective speculative execution on accelerators. Software frames consist of three components (see Figure 8) the *frame*, a block of operations to run on the accelerator, the *guards* consisting of the control flow operations, and the *undo log* that captures values in locations modified by the frame to revert in case of speculation failure. All the branches within a frame (see ♢ `W>10` in Figure 8) are converted to guards. The compiler is permitted to move instructions within the frame. When multiple paths are merged to create a Braid (e.g., paths P and P' diverging at `p==0`), then the frame introduces multiple flows of control; we rely on non-speculative predication [17] being available in the accelerator. When a guard is triggered during a frame's execution, the externally visible state has to be reverted. Note that no architectural state is shared between the frame and OOO processor; live values and memory operations are the only form of communication to and from the frame. *NEEDLE* implements the rollback using a software undo log populated by instrumenting stores.

When to invoke a BL-Path accelerator? : As program execution approaches the entry basic block for the frame (see Figure 8) it has to determine whether to invoke the frame on the accelerator or to run on the host. Predicting that the accelerated path may actually fail due to a guard failure. This is challenging in BL-Paths since they may include multiple guards, and if any fail, the entire frame must be rolled back. This issue is not as critical to *Braids* as they merge paths and reduce guards. To resolve this we use an accelerator invocation history table that maintains information on program branch history prior to the accelerator path and determines whether the BL-Path accelerator should be invoked. In our suite, 9 applications always invoked the accelerator.

VI. Evaluation

We have developed a cycle accurate simulator that models the host cores, the accelerator and data movement. The host OOO core pipeline is modelled in detail using macsim [43]. We assume that the accelerator is uncore and transfers data to the OOO core via the L2 cache. We model a CGRA fabric similar to prior work [16], [33]. The CGRAs we model are capable of issuing memory operations and are cache coherent. We model the memory operations in detail. The OOO assumes a perfect branch predictor, but the accelerator simulation models guard failures and rollback overheads to obtain a conservative estimate of speedup.

Table V
SYSTEM PARAMETERS

Host Core	1 GHz, Embedded. 4-way OOO, 96 entry ROB, 6 ALU, 2 FPU, INT RF (64 entries), FP RF (64 entries)
L1 En.	64K 4-way D-Cache, 2 cycles. LLC NUCA 8 banks, 20 cycles, MESI. Mcpat [23]; ARM 1Ghz Template.
Coarse-Grained Reconfigurable Array (CGRA)	
16×8 function units. 16 cycle reconfig. Energy Parameters (Dynamic) Network (12 pJ/switch+link), Function units (8 pJ/INT, 25pJ/FPU), 5pJ latch	

A. Performance

NEEDLE automatically identifies and offloads coarse-grained Braids to achieve a average performance improvement of 33% (max: 68%) across 29 applications.

We evaluate the performance of coarse-grained offload regions (BL-Paths and Braids) that have been *automatically curated* from large workloads using *NEEDLE*. Figure 9 shows the improvement in performance (% reduction in cycle count) for the highest ranked BL-Path and the highest ranked Braid. For the BL-Path, we quantify the performance of a) an Oracle predictor and b) a branch pattern based predictor (see § V). The precision of the predictor is displayed on the Y-scale for clarity and brevity.

The performance of the BL-Path offload is a tradeoff between the expected benefit of offload versus penalty of rollback on a guard failure. The penalty incurred in terms of performance includes the cycles spent in the accelerator as well the re-execution of the offload on the host. The performance benefit gleaned from mining more dataflow parallelism and eliding certain operations (e.g., bit manipulation) may be squandered by overly greedy invocation.

Furthermore, many workloads have a small margin for error due to the constrained nature of their dataflow graphs. In this work, we explore the bounds of offload potential by assuming a) guard failure detection only at the end of the accelerator invocation and b) CPU re-execution of a failed BL-Path.

Overall, for offloading BL-Paths we see a mean performance improvement of ≃24% across 24 applications. Five paths suffer from performance degradation, with an average of 7%. We discuss the results presented in Figure 9 with respect to their workload characteristics.

High Potential (Predictable and High ILP): ① Workloads with high ILP and coarse offload regions (e.g., 470.lbm, ferret, swaptions and sar-pfa-interp1) show significant performance improvement (up to 68%). While some workloads may be complex (e.g. swaptions with 11K paths), they demonstrate regular predictable behaviour (avg precision 98%) that translated to large gains as almost no work is wasted by wrong path rollback. Other applications such as 179.art, 197.parser are predictable, though they have lesser potential due to the nature of the computation being inherently sequential.

Low margin for error: ② 403.gcc has no ILP that the accelerator can take advantage of to improve performance. Thus, the Oracle predictor does not improve performance, and the branch history predictor degrades performance with wasted rollback being executed on the CPU. 175.vpr suffers from a similar problem due to the offloaded region being only 7 operations in size; there is no performance benefit of the accelerator. Though it is highly predictable (97%), the rollbacks for the 3% of failed executions contribute to a net 2.2% degradation.

Pathological unpredictability: ③ Due to a combination of data dependent loop branches and aggressive loop unrolling (4×) freqmine, bodytrack and blackscholes degrade performance, as the branch history patterns are insufficient. One possible approach to mitigate the issue would be to use loop fission to segregate unrolled iterations where the loop bounds are determined by data dependent values.

For Braids, we observe a mean 33% performance improvement. Note, there is low potential for degradation, as Braids have fewer guards than paths and include control flow in the offload via if-conversion. In all but one workload (sar-pfa-interp1), the highest ranked Braid provides equal or greater performance than a BL-Path with the Oracle predictor. In this workload, the BL-Path and Braid target different code regions with varying ILP.

Apples to Oranges: ④ sar-pfa-interp1 is one of the 3 workloads where the highest ranked BL-Path is not part of the highest ranked Braid. This implies that the coverage of lower ranked BL-Paths contribute to an overall higher ranked Braid. In this case, the Braid has more than 2× the number of operations and provides a higher overall energy reduction (88% vs 79%) for the BL-Path. *NEEDLE* provides a systematic approach to study offload granularity.

B. Energy Evaluation

***NEEDLE* constructed Braids in a programmer independent, automated fashion which reduced the energy consumption by 20%.** Figure 10 shows the net energy improvement for offloading Braids. While the performance improvements that can be obtained from coarse-grained offloading depend on the criticality of the dataflow graph, energy consumption can be reduced on a per operation basis due to the elimination of a processor front-end. We present only Braids in this section due to their higher performance and simplicity. We present net reduction for the entire workload region (hottest function) in contrast to the Braid only to highlight the utility of our tool.

The coverage of the Braid is indicated on top of each bar. The reduction in energy consumption is commensurate with the coverage apart from a couple of application pairs.

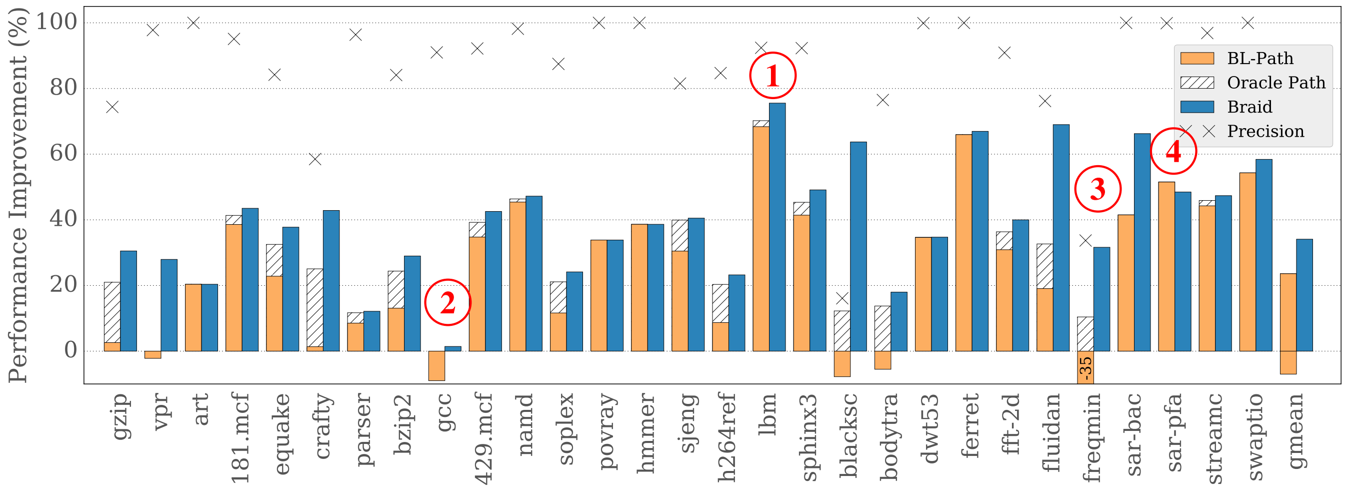


Figure 9. Performance Improvement

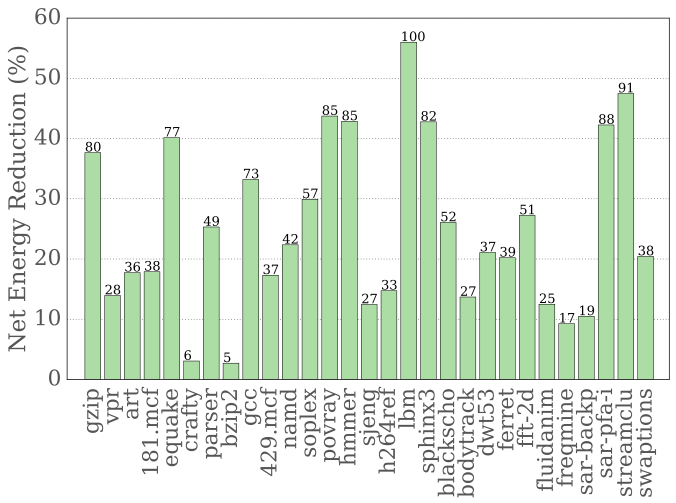


Figure 10. Net Energy Reduction for Braid

For 453.povray and 456.hmmer, the coverage is the same however the net energy reduction for 456.hmmer is lower. ferret has higher coverage than dwt53 yet has a lesser impact on energy reduction. Both pairs are due to the increased dataflow graph dependencies ($2\times$ in ferret $\leftrightarrow$ dwt53) and ($1.4\times$ in 453.povray $\leftrightarrow$ 456.hmmer).

While 401.bzip2 comprises $> 3K$ Braids, we find a single Braid (5% coverage) reduces energy by 2.7% overall. Similarly, just one Braid for 186.crafty reduces energy by $2\times$. Both these Braids offer much larger energy reduction per coverage than other workloads. The other workloads that demonstrate similar behavior are fluidanimate, freqmine and sar-backprojection. In general, floating point workloads enjoy larger reductions in energy consumption due to reduced cost of floating point operations on the spatial fabric as well as simpler control flow structure.

HLS for *NEEDLE* identified Braids: *NEEDLE* also enables high level synthesis tools [9] to target irregular workloads. We target a Altera Cyclone V SoC Processor/FPGA that combines dual ARM cores with a tightly coupled FPGA fabric. We synthesize both BL-Paths and Braids. Data can be moved between the ARM cores and the FPGA through the cache coherent AXI bus, and the memory maps permit us to share up to 1GB of coherence space between the core and the FPGA. Our backend RTL generator includes support for only a subset of the LLVM IR (v3.8). We use this infrastructure for functional testing of the hardware accelerators and area tradeoff analysis.

We synthesize RTL for 22 workloads from *NEEDLE* identified hot Braids. We target an Altera Cyclone V SoC device and find that for all but four workloads, the Adaptive Logic Modules used is less than 20% (total $\simeq$85K). For these workloads the average is 38%, max utilization is for 470.lbm (72%) where double precision floating operations are used. Modelsim simulations for power revealed that apart from three workloads all others consumed 5–60mW. The remaining three consumed 80mW, 175mW and 305mW for 444.namd, 470.lbm and swaptions respectively.

VII. Conclusion

NEEDLE is a automated tool chain that enables precise profiling, selection, and construction of "accelerator-friendly" regions. *NEEDLE* is independent of accelerator architecture and released as free and open source software. We introduce a new program path abstraction, "Braids", that merges paths with many common basic blocks to help increase accelerator code coverage without impacting the hardware complexity and energy efficiency. Finally, we use Braids to enable energy efficient software speculation on accelerators. Overall, we enable offload of irregular workloads to accelerators and achieve a 34% improvement in performance and 20% reduction in energy.

References

[1] J. R. Allen, K. Kennedy, C. Porterfield, and J. Warren. Conversion of control dependence to data dependence. In *PROC. of the 10th POPL*, 1983.

[2] 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.

[3] 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.

[4] T. Ball and J. R. Larus. Efficient Path Profiling. In *PROC of the 1996 MICRO*, 1996.

[5] 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. *PROC of the 18th HPCA*, pages 1–12, 2012.

[6] 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. *PROC. of the HPCA*, pages 1–12, 2012.

[7] D. Bryant. Disrupting the data center to create the digital services economy. 2014.

[8] M. Budiu, G. Venkataramani, T. Chelcea, and S. C. Goldstein. Spatial computation. In *PROC of the 11th ASPLOS*, 2004.

[9] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, J. H. Anderson, S. Brown, and T. Czajkowski. Legup: high-level synthesis for fpga-based processor/accelerator systems. In *PROC. of the 39th FPGA*, pages 33–36. ACM, 2011.

[10] 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.

[11] M. U. Farooq, L. John, and M. F. Jacome. Compiler Controlled Speculation for Power Aware ILP Extraction in Dataflow Architectures. In *Proc. of 4th HiPEAC*, 2008.

[12] E. Fluhr, S. Baumgartner, D. Boerstler, J. Bulzacchelli, T. Diemoz, D. Dreps, G. English, J. Friedrich, A. Gattiker, T. Gloekler, et al. The 12-Core POWER8 Processor with 7.6 Tb/s IO bandwidth, Integrated Voltage Regulation, and Resonant Clocking. 2015.

[13] C. Frericks, R. Cofell, and K. Sankaralingam. Performance evaluation of a DySER FPGA prototype system spanning

the compiler, microarchitecture, and hardware implementation. 2015.

[14] M. Gebhart, B. A. Maher, K. E. Coons, J. Diamond, P. Gratz, M. Marino, N. Ranganathan, B. Robatmili, A. Smith, J. Burrill, S. W. Keckler, D. Burger, and K. S. McKinley. An evaluation of the TRIPS computer system. In *PROC of the 14th ASPLOS*, 2009.

[15] C. González-Álvarez, J. B. Sartor, C. Álvarez, D. Jiménez-González, and L. Eeckhout. Automatic design of domain-specific instructions for low-power processors. *In PROC. of ASAP*, pages 1–8, 2015.

[16] V. Govindaraju, C.-H. Ho, T. Nowatzki, J. Chhugani, N. Satish, K. Sankaralingam, and C. Kim. Dyser: Unifying functionality and parallelism specialization for energy-efficient computing. *IEEE Micro*, 32(5):0038–51, 2012.

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

[18] 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.

[19] R. Hameed, W. Qadeer, M. Wachs, O. Azizi, A. Solomatnikov, B. C. Lee, S. Richardson, C. Kozyrakis, and M. Horowitz. Understanding sources of inefficiency in general-purpose chips. In *PROC of the 37th ISCA*, 2010.

[20] M. Hill and C. Kozyrakis. Advancing computer systems without technology progress. In *DARPA/ISAT Workshop*, 2012.

[21] 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 *PROC. of the PLDI*, 2012.

[22] S. Kumar. Needle on github.

[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] F. Liu, H. Ahn, S. R. Beard, T. Oh, and D. I. August. Dynaspam: dynamic spatial architecture mapping using out of order instruction schedules. In *PROC. of the 42nd ISCA*, pages 541–553, 2015.

[25] 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*, 7(1-2), May 1993.

[26] S. A. Mahlke, R. E. Hank, J. E. McCormick, D. I. August, and W.-m. W. Hwu. A comparison of full and partial predicated execution support for ILP processors. In *PROC of the 22nd ISCA*, 1995.

[27] 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.

[28] 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.

[29] 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*, 7:229–248, 1993.

[30] T. Nowatzki, V. Gangadhar, and K. Sankaralingam. Exploring the potential of heterogeneous von neumann/dataflow execution models. In *PROC of the 42nd ISCA*, New York, New York, USA, June 2015.

[31] T. Nowatzki, V. Govindaraju, and K. Sankaralingam. A Graph-Based Program Representation for Analyzing Hardware Specialization Approaches. *IEEE Computer Architecture Letters*, 2015.

[32] A. Parashar, M. Pellauer, M. Adler, B. Ahsan, N. Crago, D. Lustig, V. Pavlov, A. Zhai, M. Gambhir, A. Jaleel, R. Allmon, R. Rayess, S. Maresh, and J. Emer. Triggered instructions: a control paradigm for spatially-programmed architectures. In *PROC of the 40th ISCA*, Apr. 2013.

[33] H. Park, Y. Park, and S. Mahlke. Polymorphic pipeline array: a flexible multicore accelerator with virtualized execution for mobile multimedia applications. In *PROC of the 42nd MICRO*, 2009.

[34] Y. Park, H. Park, and S. Mahlke. *CGRA Express: Accelerating Execution Using Dynamic Operation Fusion*. CASES '09. 2009.

[35] D. N. Pnevmatikatos and G. S. Sohi. Guarded execution and branch prediction in dynamic ILP processors. In *PROC of the 21st ISCA*, 1994.

[36] A. Putnam, A. M. Caulfield, E. S. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, G. Prashanth, G. Jan, G. Michael, H. S. Hauck, S. Heil, A. Hormati, J.-Y. Kim, S. Lanka, J. Larus, E. Peterson, S. Pope, A. Smith, J. Thong, P. Yi, and X. D. Burger. A reconfigurable fabric for accelerating large-scale datacenter services. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, 2014.

[37] W. Qadeer, R. Hameed, O. Shacham, P. Venkatesan, C. Kozyrakis, and M. Horowitz. Convolution Engine: Balancing Efficiency & Flexibility in Specialized Computing. *PROC of the 40th ISCA*, pages 1–12, Apr. 2013.

[38] B. Reagen, R. Adolf, S. Y. Shao, G.-Y. Wei, and D. Brooks. Machsuite: Benchmarks for accelerator design and customized architectures. In *IEEE International Symposium on Workload Characterization (IISWC)*, 2014.

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

[40] Y. S. Shao, B. Reagen, G.-Y. Wei, and D. M. Brooks. Aladdin: A pre-RTL, power-performance accelerator simulator enabling large design space exploration of customized architectures. In *Proc. of the 41st ISCA*, pages 97–108, 2014.

[41] H. Singh, M.-H. Lee, G. Lu, F. J. Kurdahi, N. Bagherzadeh, and E. M. C. Filho. MorphoSys: an integrated reconfigurable system for data-parallel and computation-intensive applications. *IEEE Transactions on Computers*, 49(5):465–481, May 2000.

[42] A. Smith, J. Gibson, B. A. Maher, N. Nethercote, B. Yoder, D. Burger, K. S. McKinley, and J. H. Burrill. Compiling for EDGE Architectures. *PROC. of the CGO*, 2006.

[43] G. Tech. Macsim : Simulator for heterogeneous architecture - https://code.google.com/p/macsim/.

[44] G. Venkatesh, J. Sampson, N. Goulding, S. Garcia, V. Bryksin, J. Lugo-Martinez, S. Swanson, and M. B. Taylor. Conservation cores: reducing the energy of mature computations. In *PROC of the 15th ASPLOS*, 2010.

[45] G. Venkatesh, J. Sampson, N. Goulding-Hotta, S. K. Venkata, M. B. Taylor, and S. Swanson. Qscores: Trading dark silicon for scalable energy efficiency with quasi-specific cores. In *PROC. of the 44th MICRO*, 2011.

[46] M. A. Watkins, T. Nowatzki, and A. Carno. Software transparent dynamic binary translation for coarse-grain reconfigurable architectures. In *PROC. of the 21st HPCA*, 2016.