SPEC-AX and PARSEC-AX: Extracting Accelerator Benchmarks from Microprocessor Benchmarks

Snehasish Kumar, William N. Sumner, Arrvindh Shriraman
School of Computing Science
Simon Fraser University
{ska124, wsumner , ashriram}@cs.sfu.ca

Abstract—**The end of Dennard Scaling has necessitated research into the adoption of specialized architectures for offloading specific code regions in applications. Recent works in accelerator architectures have chosen diverse workloads and even diverse code regions (within the same workload) to highlight the efficacy of specific accelerator architectures. However this makes it challenging to evaluate the power/performance benefits of each accelerator. It is unclear in the era of specialization whether it will be feasible to standardize a new set of kernels across different architectural ideas. We present an alternative vision where we identify and prepare "acceleratable" code regions from existing CPU-based benchmark suites that are widely used. We identify acceleratable paths by leveraging program analysis [1] to precisely identify directed acyclic paths that are frequently executed. We reconstruct the paths into a separate function within the original binary and demarcate the accelerator region to enable microarchitecture independent analysis and enable precise profiling when executing the program on an architecture simulator or instrumentation tool (e.g., Intel Pin). We extract "accelerator" offload targets from frequently executed paths for 29 workloads across SPEC2000, SPECCPU2006, PARSEC and PERFECT benchmark suites and demonstrate that characterization along paths is more precise than characterization at coarser granularities in prior work. Overall, we analyze 356K paths across 29 workloads and present statistics for the top 5 paths identified for offload in each application. We have also generated a workload suite with the acceleratable code paths to help computer architecture researchers.**

I. Introduction

A central tenet of the modern accelerator proposals is to split up programs into multiple phases and use a specialized the architecture to target the behavior of each phase (e.g., SIMD, CGRA [12], specialized instructions [8]). With the end of Dennard scaling, improving the performance of general-purpose processors has proven to be quite challenging, placing more emphasis on architects. While it is clear that customized hardware accelerators exploiting specific program behaviors is a promising way forward, it is not clear what the particular accelerator microarchitecture is and how can we compare alternative microarchitectures that target the same behavior. We have made great strides in cases where the hardware needs to target an already mature application domain (e.g., SIMD or GPUs).

The objective in accelerators is typically to design a fixed-function or programmable hardware that provides the necessary support for a given program behavior with the lowest possible overhead (area/power). In contrast, a general-purpose processor tries to maximize performance across all applications for a given cost. It is imperative that the computer architects have access to the specific code regions within existing target applications so that accelerators can be designed with confidence. It is imperative for designers to understand not just the statistical characteristics and microarchitectural behavior [9] of the specific applications but also the precise functionality and semantics of code when designing the custom hardware. By design, accelerators are expected to provide functionality and performance only for a narrow phase of the application.

However, many critical real world applications were developed for CPUs and do not have explicitly marked phases of the program on which computer architects and designers can focus the accelerator design effort. The absence of such real world workloads makes it extremely challenging to reliably develop accelerators hoping that they will be used in existing or future applications. Unfortunately, there have not been good benchmark suites. Current accelerator-specific suites [20] are essentially important kernels from libraries in mature application domains, but real world workloads are significantly more complex. By design, different fixed-domain and fixed-function accelerator proposals tend to pick algorithms and kernels from a specific application domain (e.g., machine learning or databases). It is not clear how to compare other accelerator architectures that may target the same code region or what code regions within a workload different accelerators should even target. We suspect that in the workloads that prior researchers have targeted [22], a specific behavior dominated entire execution of the benchmark.

Benchmarking is a key tool for assessing computer systems. A core benefit of benchmarks is to enable comparing design alternatives during research or development and evaluating power/performance tradeoffs. Inflection points in computing systems (e.g., multicore, cloud computing) have resulted in new benchmark suites (e.g., PARSEC [6], CLOUDSUITE [10]). The pitfall and limitation is that these benchmarks may not be representative of real-life applications and may be very different from the application(s) of interest. An alternative would be to use real-life applications of interest. Unfortunately, real-life applications are very often challenging to set up with the need for mature compiler, operating system and library stacks (typically not available with hardware accelerators). This introduces a chicken-and-egg problem: designing accelerators suitable for applications requires the applications to convey

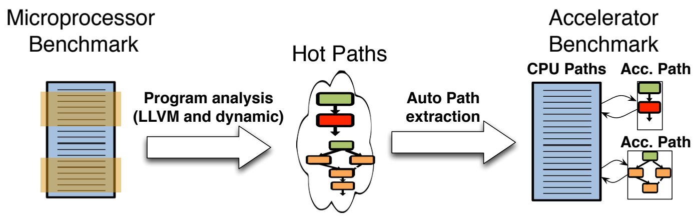


Fig. 1. Using program analysis to demarcate and extract code paths [1] for accelerators within CPU programs.

precisely what function needs to be accelerated in the first place. We take an alternative approach. Instead of developing a new benchmark suite for hardware acceleration, we highlight specific code regions within existing applications that accelerators should target.

The objective of this paper is to explore accelerators in existing CPU-based applications and make it possible for architects and designers to rapidly explore behaviors to specialize and accelerate. We ensure that the accelerators developed for the demarcated regions within each application can be directly deployed within the original application. In order to achieve this we have employed precise analysis of the execution paths [1] and extracted the frequently executed path into a separate function within the original program. Converse to prior work that extracted only key performance characteristics, we extract the frequently executed code region paths that an accelerator should target. We have extracted the acceleratable program paths embedded amongst many other unimportant or unacceleratable code regions into functions that accelerator compilers can target or use for simulation studies (e.g., Pin instrumentation [16]). We demonstrate that extracting such frequently executed paths in many cases requires carefully navigating across the control flow and precisely characterizing the biases of the control flow.

The approach that we propose overcomes an important shortcoming of the existing benchmarks – benchmarks only seek to retain the memory access patterns and control flow behavior similar to the workload they represent [15]. Similarity is typically characterized using statistics such as branch biases or cache miss rates, which may suffice for studying microarchitectural resource characteristics (e.g., branch prediction or cache architectures). While benchmarks have sufficed to study general-purpose microprocessor characteristics, they are too imprecise to indicate the specific code behavior that should be accelerated. We have extracted the specific code paths and ensure that our extracted paths i) replicate the functional semantics of the original application region ii) include the control flow of the original program, and iii) mimic the memory access behaviour of the original program. Figure 1 illustrates our overall approach.

Overall we make the following contributions:

1) We have developed a robust LLVM-based compiler infrastructure that precisely and automatically identifies the frequently executed code paths in an application. (§ II)
2) We comprehensively study the acceleration characteristics (e.g., operation count, type, memory behavior) demonstrated along program paths [1] and precisely characterize the behavior (§ III).
3) We provide a derived benchmark suite with code paths demarcated for acceleration and extracted and prepared for analysis. These accelerator-independent code paths can be used by researchers within existing simulation (e.g., GEM5) and analysis tools (e.g., Intel Pin) (§ V)

II. Motivation & Methodology

a) HLS Benchmarks vs Accelerator Benchmarks: : It is important to clarify the intent of accelerator benchmark suites (like this paper) against benchmarks used for improving high level synthesis (HLS) toolchains. Examples of HLS suites include Machsuite and CHSTONE. HLS benchmarks are a lot simpler than CPU benchmarks and are kernelized to target HLS toolchains (e.g., Xilinx's Vivado). CPU benchmarks are rarely written as a collection of kernels and hence HLS benchmarks are not representative of CPU benchmarks. Compared to HLS benchmarks, accelerator paths in SPEC and PARSEC have significantly more complex code (i.e., more branching behavior), are intermingled with many other code paths (i.e., applications are not dominated by an obvious set of kernels), rarely have regular strided memory access patterns (i.e., many levels of memory locality behavior), and have diverse compute and memory characteristics. It is imperative that studies designing hardware accelerators [23] for HLS benchmarks be aware that their conclusions might not extend to existing CPU workloads such as PARSEC and SPEC. Our workloads are intended for those seeking to design accelerators that are useful for existing CPU workloads, whether they pass through HLS toolchains or not (infact due to the complexity of our paths only 5 of our workloads pass through an existing HLS toolchain [7]).

b) Methodology: Workloads often exhibit varied behavior internally. For instance, a program may have different phases of behavior representing initialization, computation, or cleanup. These phases perform different tasks and, as a result, exhibit different characteristics. However, analyzing an entire workload at once blends the characteristics of these different tasks together, obscuring the patterns in behavior of any one specific task. When exploring which behaviors in a workload to accelerate, these blended results may make it harder to tease out the characteristics of a particular program segment that capture desirable or undesirable behavior.

At a fine granularity of program segment, different acyclic paths in a program may exhibit different characteristics than other paths. A *path* is simply a sequence of instructions in a program. An acyclic path is a sequence of instructions that starts either at the beginning of a program or immediately after a back edge in a control flow graph and terminates at the end of a program or at the next back edge. Intuitively, acyclic paths divide the behavior of a program into loop free segments. For example, in Fig. 2, 1234 is an acyclic path that represents entering a loop starting at 2 but not revisiting 2. The acyclic path 234 represents a single iteration of the

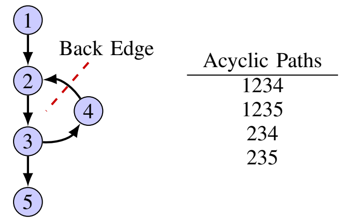


Fig. 2. Acyclic paths in a control flow graph

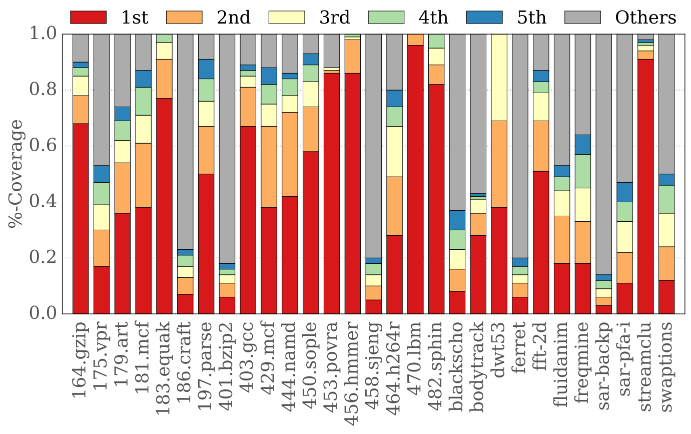


Fig. 3. Path Bias

loop starting at 2, while the acyclic path 235 exits the loop. A function in a program comprises its constituent acyclic paths as well as any back edges that connect them. Thus, characterizing a workload's behavior at the function granularity will combine characteristics of all constituent paths within that function.

Some paths may be more beneficial to accelerate than others. For instance, when most of the computation in a workload happens along one path, it may be more fruitful to accelerate that path than an alternative. Indeed, we have found that the real world behavior of workloads can be highly biased toward some acyclic paths over others. Fig. 3 examines the relative coverage (in terms of dynamically executed instructions) of different paths for the hottest functions in workloads as determined by `gperftools` [11]. It presents the relative coverage of a workload provided by the five most frequently executed paths in the selected function vs the coverage of the workload provided by all other paths in the function combined. Note that for most workloads, these five hottest acyclic paths are sufficient to cover the majority of workload's behavior. In many cases, the hottest path alone dominates the coverage. Accelerating those particular paths can thus be more beneficial than accelerating others. However, characterizing a workload at a coarser granularity, such as an entire function or loop body, will blend the characteristics across paths, once again potentially obscuring information that may help in making acceleration decisions. To overcome this problem, workloads can be characterized along acyclic paths in order to capture program behaviors within these fine grained program segments.

To examine the impact of characterizing paths of workloads, we first identify the most frequently executed acyclic paths within each workload. We then statically reconstruct each of these paths into independent functions and collect machine independent characteristics for each selected path in each workload. This section discusses the benchmarks that we used as well as our approach for selecting, reconstructing, and characterizing program paths.

A. Selecting Paths to Characterize

Identifying frequently executed paths is an important part of many analyses for both architecture and for software. The classic approach to addressing this problem is to use the efficient path profiling technique developed by Ball and Larus [1]. This technique instruments a program to produce a dynamic profile as it runs. The instrumentation process first decomposes the control flow graph of a function into acyclic paths and assigns each path a unique integer id in the range from 0 to the total number of paths. Next, the program is instrumented so that the id for a path is computed as that path executes within the program, and the count or frequency of a path is incremented once the end of the path is reached. The end result of running an instrumented program is a count of how many times each acyclic path through a function is executed.

Efficient path profiling provides the foundation for our approach to identifying which paths inside a workload to characterize. For each workload in our benchmark suite, we select the 5 most frequently executed acyclic paths through the workload. Note, however, that the default efficient path profiling algorithm does not identify the frequencies of paths through an entire program, rather, it identifies the frequencies of paths through individual functions. Thus, we must adapt path profiling in order to profile acyclic paths at the program level.

In order to profile acyclic paths at the program level, we merge the control flow graphs of the entire program into a single function. We perform this by running an aggressive inlining pass on the LLVM intermediate representation (*IR*) of a workload. This aggressive optimization performs function inlining at every possible call site within the IR.

With inlining completed, efficient path profiling again enables us to identify the most frequently executed paths in the entire program. However, inlining introduces additional engineering burdens that must first be addressed. In particular, the number of acyclic paths through an entire program is larger than the number of paths through just one function within a program. As a result of inlining, the total number of paths need not be representable as a single integer during the profiling process. Column C1 in Table I shows the number of bits required to represent all the paths for a particular workload.

After performing path profiling on the fully inlined version of the program, we select the top five most frequently executed paths from each workload. Note that recursive function calls and calls through function pointers cannot necessarily be inlined. These constructs partition the program into disjoint functions after aggressive inlining. In these cases, we select the hottest of the remaining functions using `gperftools`.

B. Extracting identified paths

After identifying the most frequently executed paths inside each workload, we extracted each such path into its own function for easier, more isolated study. For each path, we created a new function containing the same sequence of instructions as in the original path. Note that a branch instruction in the middle of the path can force program execution to deviate from the path once it has started. When this happens, the function returns early, and the original version of the program is executed. Thus, we call these exit guards. All incoming dependencies from live-ins inside the path are hoisted to arguments of the function, while all outgoing dependencies are returned through a struct when the function completes. All store instructions are recorded to an undo buffer that is replayed when the path exits early. After extracting each path into its own function, we once again run `-O2` optimizations to remove any unnecessary operations and simplify the path specific behavior. By extracting these paths into their own functions, we have created durable artifacts that may be reused by other analyses.

C. Metrics & ISA-independence

We base our workload metrics upon the prior work by Shao [21]. In particular, we examine the unique opcodes, the memory address entropy, and the number of guards or unique branches. We also extend their metrics into analogues at the path granularity. This includes path predictability, an analogue of branch entropy, as well as the average number of read and write operations across extracted paths for a workload, which together provide an upper bound to the memory footprint. Finally, we add metrics that are more relevant for analyzing acyclic paths. This includes the number of live in and live out values to the path, the number of ϕ operations removed when extracting the path from the original control flow graph, and the total number of static instructions in the path. More details are provided in § III-B.

Our analysis operates on the level of LLVM IR. By utilizing LLVM IR as our representation for characterization, we are able to draw conclusions that better reflect the intrinsic semantics of the original program. Prior work has shown this to be highly desirable [21].

D. Characterizing at the Path Level

Static characteristics of the extracted paths are computed directly from their corresponding functions. Note that applying optimizations again after extracting each path into its own function produces characteristics that are more reflective of that particular path's behavior. Any computation used only on branches that exit from the path is removed, and the remaining computation is simplified to more accurately reflect the behavior of just the path of interest.

Dynamic characteristics of the path, namely the addresses of loads and stores to heap allocated memory, are computed by re-executing the entire workload with the path of interest outlined into its respective extracted function. The addresses of heap accesses are recorded using Pin for further characterization via, e.g., memory entropy. Stack accesses are ignored, as they reflect more architectural dependent characteristics rather than intrinsic behaviors of the workloads of interest.

E. Benchmarks

We include 29 workloads from SPEC2000, SPEC2006, PERFECT [2] and PARSEC [5]. [1] All benchmarks were compiled with clang version 3.8 in order to generate LLVM bitcode for instrumentation and analysis. We perform aggressive loop unrolling ($4\times$) with an increased threshold ($2\times$) and allow partial unrolling. Both before and after instrumentation, all workloads were optimized at the level of `-O2` with vectorization disabled. Executable versions of the extracted workloads were then compiled for X86-64 using LLVM 3.8.

III. CHARACTERIZATION

This section highlights the disparate behaviour of workloads along frequently executed acyclic paths. Our approach is in contrast to prior work [21], [26], which examines workloads as a whole or at a function granularity. We find that considering paths as the granularity for characterization yields insightful information for specialization. We describe our methodology in § II.

A. Making a case for Path-based Acceleration

The key to an effective offload abstraction is that it must concisely capture varied dynamic phase behavior exhibited by a workload. For instance, a program may have phases of behavior representing initialization, computation, or cleanup that exhibit different execution characteristics. Analyzing an entire workload at once blends the characteristics of these different tasks together, obscuring the patterns in behavior that should be accelerated. When exploring which behaviors in a workload to accelerate, these blended results may make it harder to tease out the characteristics of a particular program segment that capture desirable or undesirable behavior. We show that paths executed by even a single program exhibit diverse characteristics and are a natural fit for specialization. A path is simply a sequence of dynamic instructions in a program. Intuitively, acyclic paths divide the behavior of a program into loop free segments. A function in a program comprises its constituent acyclic paths as well as any back edges that connect them. Thus, characterizing a workloads behavior at the function granularity will combine characteristics of all constituent paths, obscuring overall behavior. Furthermore, some paths may be more beneficial to accelerate than others. For instance, when most of the computation in a workload happens along one path, it may be more fruitful to accelerate that path. Indeed, we have found that the behavior of real workloads is highly biased with only a few hot paths.

[1]We drop benchmark programs where the selected function contains language features unsuitable for an accelerator: e.g.,setjmp and longjmp in 471.omnetpp and C++ exceptions 447.dealII in 483.xalancbmk.

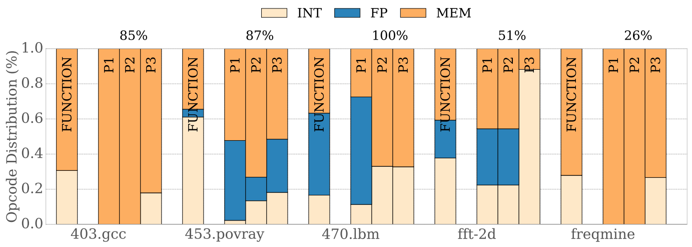


Fig. 5. Benefits of Path-Based Execution. We have only shown a few workloads due to lack of space. Opcode histogram of paths within a function; % indicates exec coverage.

The histogram in the Figure 5 demonstrates that the granularity of program analysis and offload abstraction fundamentally biases what to include on the hardware accelerator. The opcode distributions are shown for the top 3 hot paths in the most important function the program, and the % indicates the total execution coverage achieved by the paths. An example that motivates our approach is 470.lbm. The function breakdown shows an overall bias for FP (60% of dynamic ops). However, we see that the only one path executes floating point instructions. The remaining paths are dominated by MEM operations, and they have no FP operations at all! An accelerator designed to offload each path separately can be customized to support only the operations in that path. Another interesting observation is that while 453.povray is an INT-heavy function (60%+ operations), the hottest paths cover 87% of the dynamic execution yet consist less than 20% of INT operations. Thus, some cold path with INT ops is biasing the overall function. Finally, we highlight freqmine and gcc as cases where the overall function can be easily segregated into paths that access memory and paths that compute, which permits the synthesis of fully decoupled specialized accelerators. Another benefit of path-based regions is saving of wasted work. Typical program regions tend to have multiple execution paths due to control flow and the relative hotness of these paths is exhibited only by dynamic execution profiles. Current HLS tools use a static approach to acceleration region formation and conservatively offload multiple paths.

B. Characteristics Summary

Table I presents key characteristics for the workloads we study. Column C1 indicates the number of bits required to encode all the static paths that a workload may execute. We see a large variance across workloads depending on their nature. Some of the more complex workloads we study are swaptions and 186.crafty with 73 and 63 bits required to enumerate all paths. Path explosion is described in more detail in § II. Often, floating point workloads demonstrate less complex structure. Workloads such as 470.lbm, 183.equake and 482.sphinx3 all require fewer than 5 bits to enumerate all paths.In comparison to the potential number of paths in a workload, the actual number of unique paths executed is often low. Only 11 workloads have more than 1000 paths executed during program execution. 401.bzip2 has the largest number of paths executed with over 72K. The median number of paths executed is 250. Across the workloads we study, there exists path bias, i.e few paths executed far more frequently than others (see Figure 3).

Columns C3-C8 in Table I provide the maximum value of a particular characteristic across the five most frequent paths of a workload. This data combined with the normalized visualization presented in Figure 6 allows the reader to derive the absolute values for each of the 143 paths (across 29 workloads) presented. The observations are discussed in a workload centric manner in § IV. Herein, we discuss the path characteristics across workloads and their implications on accelerator synthesis.

Path Length and Opcode Mix : Column C3 in Table I shows the size of the largest path for each workload. The largest path overall was from 183.equake with 962 IR instructions. The median size of the largest path from each workload across the suite was 232 instructions. 7 out of 29 workloads have fewer than 100 operations, and four workloads have paths with more than 500 operations.

Prior work such as BERET [13], has sought to accelerate "Superblock" regions. Such characterization is often limited by predetermined hardware constraints of the accelerator. Our path based characterization yields different results as we do

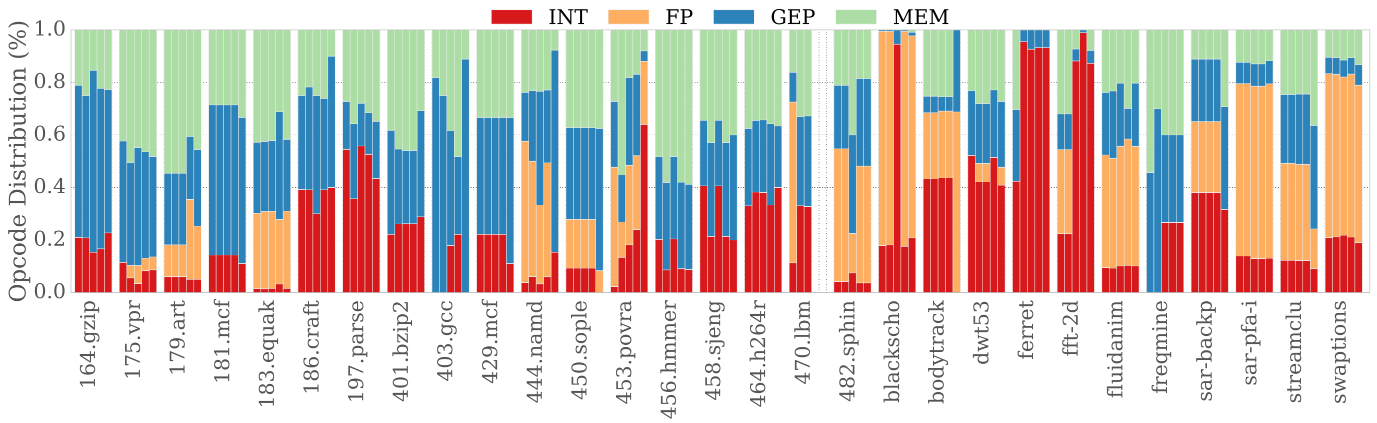


Fig. 4. Opcode Distribution. The 5 bars for each workload represent the top-5 hot paths (L-R), GEP=pointer access.

TABLE I
WORKLOAD CHARACTERISTICS

C1 : log2(NumPaths) **C2** : Exe. Paths (MAX_5) **C3** : Ins. **C4** Cov. **C5** : Guards **C6** : Phi Nodes Removed
C7 : Live Vals **C8** : Mem. Entropy ($GEOMEAN_5$) **C9** : Mem. RD **C10** : Mem. WR **C11** : Num Unique Opcodes

		C1	C2	C3	C4	C5	C6	C7	C8	C9	C10	C11
Workload	Function	Bits	Exec	Size	Cov%	◇	ϕ	V↑↓	M.S	M.↓	M.↑	{Op}
164.gzip	longest_match	10.4	813	70	59	7	11	6,6	16.0	4.2	0	11
175.vpr	try_route	79.8	5394	332	2	25	10	9,6	14.6	26.8	6.2	13.8
179.art	match	19.8	6082	174	11	4	3	3,3	10.0	31.2	4	13.2
181.mcf	price_out_impl	10.8	402	21	1	2	2	3,2	8.3	2.2	0	7.8
183.equake	smvp	4.3	13	962	53	4	8	14,11	17.3	131.4	8.4	10.6
186.crafty	EvaluatePawns	62.7	37443	67	03	14	12	11,4	6.9	4.8	0	10.6
197.parser	table_pointer	9.5	250	115	51	12	2	7,2	10.8	8.8	2.6	12.8
401.bzip2	BZ2_compressBlock	69.5	72561	784	05	71	6	9,8	18.3	100.8	4	11
403.gcc	bitmap_operation	11.9	21	100	67	6	13	9,7	10.0	4.4	2	8
429.mcf	price_out_impl	9.7	141	24	1	2	2	3,2	8.9	3.0	0	8.8
444.namd	calc_pair_energy_fullelect	14.8	249	673	44	8	16	36,16	12.0	22.6	6.6	13.4
450.soplex	vSolveUrightNoNZ	9.3	389	94	13	4	3	11,4	12.8	11.0	3.6	11.8
453.povray	All_Sphere_Intersections	17.8	33377	331	86	10	10	15,12	6.0	16.6	5.8	17.6
456.hmmer	P7Viterbi	13.8	36	490	71	8	7	20,2	16	47.2	20.2	9.2
458.sjeng	gen	34.3	46971	95	05	13	4	3,3	5.4	6.8	0.8	10.6
464.h264ref	dct_luma_16x16	26.5	88	433	19	25	25	14,5	8.3	48.0	8	16
470.lbm	LBM_performStreamCollide	2.3	2	479	96	2	1	3,2	19.6	26.0	19	10.7
482.sphinx3	vector_gautbl_eval_logs3	5.9	9	154	4	4	4	13,8	14.0	12.0	1.2	11.6
blackscholes	BlkSchlsEqEuroNoDiv	22	34	314	08	32	52	9,4	2	0.4	0	20
bodytrack	InsideError	18.8	64516	233	16	16	6	12,5	4.7	16.0	6.4	15.2
dwt53	dwt53_row_transpose	5.9	12	122	37	4	1	9,2	19.0	9.8	5.4	12.6
ferret	image_segment	19.0	31136	485	04	32	54	8,7	17.3	7.2	4.8	12.6
fft-2d	fft	27.9	46	232	24	28	5	8,1	17.2	11.2	8	14.4
fluidanimate	ComputeForces	23.1	39838	143	13	12	4	18,3	14.0	13.8	1.2	14.8
freqmine	conditional_pattern_base	8.4	133	94	13	4	3	6,8	10.2	8.0	3.6	9.4
sar-backp	sar_backprojection	77.7	4616	127	01	13	9	12,7	8.4	4.2	3.8	22.2
sar-pfa-in	sar_interp1	40	173	509	07	54	29	17,3	6.5	23.4	7.6	22.4
streamc	pgain	11.4	74	249	41	16	3	11,6	13.6	27.4	0.6	13.6
swaptions	HJM_Swaption_Blocking	72.5	11663	462	12	30	18	9,3	11.5	24.0	8	24.8

not have any preconceived notion of the specialization target. Figure 4 shows the distribution of Opcodes across the five frequent paths for each workload. We classify the opcodes as INT, FP, MEM and GEP.

GEP operations in the LLVM IR are a succinct representation of address generation logic. They define, in a platform independent manner, the operations required to generate a particular memory address prior to the access. Classifying GEPs separately allows us to quantify "work" required to fetch data independent of the actual compute on the data. Overall Figure 4 shows that across the frequent paths in a workload there may be significant differences in their opcode mix. GEP and MEM operations tend to account for a significant fraction of the work in the hot paths across workloads, on average 45% of the number of operations. Only 49 of 143 paths have more compute (INT+FP) operations than memory (GEP+MEM).

In some floating point workloads, amongst the top 5 paths, there exist paths with no floating point operations at all. Workloads such as 444.namd, 470.lbm and blackscholes have at least one or more frequent paths devoid of floating point operations.

Conversely, four of the top five paths in 175.vpr have floating point operations (5% of total) on average. Similarly for dwt53, 7% of the operations across three of the top five paths are floating point operations (primary datatype was integer – `typedef int algPixel_t`.

Another interesting observation is the presence of paths with only GEP operations or GEP and MEM operations but no compute. 470.lbm is a workload with two paths that only compute GEP expressions. One of the paths is the macro definition `SWEEP_START`. The macro is defined shown in Listing 1.

Listing 1. Macro definition – 470.lbm

```
#define SWEEP_START(x1,y1,z1,x2,y2,z2) \
    for(i = CALC_INDEX(x1, y1, z1, 0); \
        i < CALC_INDEX(x2, y2, z2, 0); \
            i += N_CELL_ENTRIES ) {
```

This particular case occurs in freqmine (2 paths) and 403.gcc as well. Column C11 in Table I shows the average number of unique IR instructions that are present in the top five paths. The number ranges from ≃8 to 25 unique IR operations across the workloads. Within workloads the variability is low. The total number of opcodes present in the IR is 64 (LLVM 3.8).

Branches, Guards and ϕ : Column C4 in Table I shows the number of conditional branches converted to guard checks for exiting the middle of a path. Guards are discussed in more detail in § II. The presented number is the maximum across the five frequent paths for each workload. The largest paths, 401.bzip2 and sar-pfa-interp1 have 71 and 54 guard checks respectively. All other workloads have 32 or fewer guards, and 13 workloads have <10 guards. The largest "guard density" (guards divided by size) we find is 22% for 183.crafty.

ϕ's in the LLVM IR are instructions that select incoming values based on the result of control flow operations. ϕ's have a direct impact on the complexity of specialization as observed in prior work [4], [18]. Reasoning about specialization along paths allows for ϕ simplification. The ϕs can be resolved since the control flow is known a priori. This is proportional to the number of branches removed (conditional and unconditional). Workload behaviour determines the number of ϕs required (and thus elided) at each branch.

The largest number of simplifications across 143 paths occur in ferret. Note that in this case there is no path correspondence with the other metrics such as the path size. The maximum number of ϕ simplifications may occur along different paths. Over 143 paths, the average number of ϕs simplified per path is 0.68 (geomean). However, it can be particularly high in some cases, with 6-8 ϕs simplified in 5 of 143 paths (workloads – 164.gzip, 183.equake, 444.namd $\times 2$, bodytrack). Note that ϕ used as induction variables are not included in this analysis.

Live Values : The live values of a path are the virtual register values that are a) used within the path (live in) and b) defined within the path and used outside (live out). Quantifying the live input and output values (Column C7) provides a notion of the overheads of data transfer into and out of the specialized unit. We present the maximum number of live values (input and output) per workload in Column C7. Memory state is treated separately and discussed in the following paragraph.

Across workloads and their top five paths, the average number of live values is 10. The maximum number of live values was observed in 444.namd (36/16 – in/out). Across benchmarks the least number of live values was observed for 181.mcf. 59 of 143 paths had fewer than 10 live values while only 1 path had more than 25 (444.namd).

Memory Access Characteristics : We present characteristics of the paths in Columns C8-C10 of Table I with respect to memory behaviour. Column C8 enumerates the maximum *memory address entropy* for the five most frequent paths per workload. This metric has been used previously [21], [27] to quantify the information content, i.e. the predictability of memory addresses. Entropy in information theory encodes the randomness of the variable. Herein, each unique location accessed is treated as a value for the variable. Lower numbers imply higher predictability. We analyze the memory address entropy for heap accesses only. Shao et al.[21] find that analyzing heap+stack addresses together provides less meaningful results. blackscholes has a pattern of reading from six arrays of same size, computing a value, and writing the computed value to a seventh array. The memory accesses are regular uniform strides and result in low memory address entropy (2). More details of the implementation are presented in § V.

Columns C9 and C10 present the average number of memory read and memory write operations across the frequent paths. The range of average number of memory reads extends from 0.4 (blackscholes) to 131.4 (183.equake) across the frequent paths in our workloads. The blackscholes benchmark from PARSEC passes input as function arguments to the hot function, thus reducing memory reads. The range of memory writes extends from 1 to 20.2 operations (for workloads with non-zero memory writes on average). For applications with zero writes (52 of 143 paths), we find paths that return live values rather than issue stores to memory. Almost all paths are "consumer" in nature, where the reads outnumber writes. One path (sar-backprojection, rank 5) has more writes than reads. Only 11 of 143 paths have more than 16 writes to memory. Note that this does not distinguish aliasing memory locations. We only comment on the number of operations per path.

TABLE II
PATH PREDICTABILITY

ID	Freq	Path	Probability
1	100	**A** B C	$100/(100+25) = 0.8$
2	25	**A** B D	$25/(100+25) = 0.2$
3	10	**F** G	$10/10 = 1.0$

IV. Path Characteristic Variability

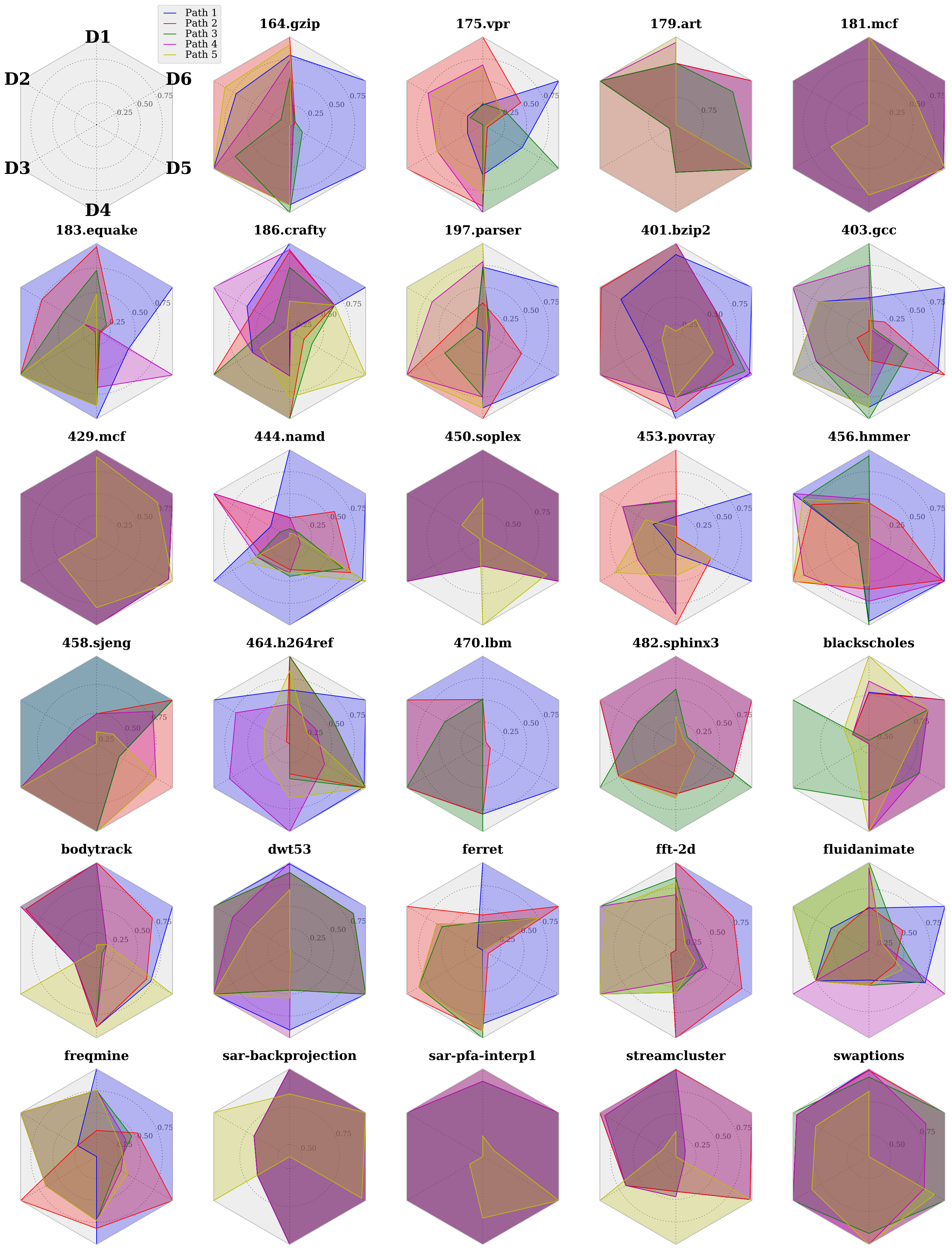
In this section, we summarize our observations across workloads. Figure 6 presents key characteristics of the five most frequent paths across workloads. We present six features to contrast paths within a workload, four of which are derived statically. Prior work [21], [18] has indicated these features are key to understanding amenability to acceleration. We observe that different program paths exhibit different characteristics.

Description : Each radar chart represents a single workload. Each outlined overlay on the chart represents a frequent path (five in total). There are six dimensions on each radar chart. In counter clockwise order, they are i) **D1** : (Norm.) Number of instructions ii) **D2** : (Norm.) Number of guards, iii) **D3** : (Norm.) Number of ϕ's simplified, iv) **D4** : (Norm.) Total number of live values v) **D5** : Predictability, vi) **D6** : Coverage.

Of the six metrics enumerated, **D1**-**D4** have been discussed previously in § III-B. The radar charts present normalized values. Absolute values per path can be derived from the max values presented in Table I (**C3**-**C7**). **D6** : Path predictability is a new metric we introduce in this section. Path predictability is the probability of following a known path given the starting basic block. Consider the contrived path profile in Table II. The predictability of each path is calculated as the execution frequency of the path divided by the sum of the frequencies of all other paths which *begin at the same basic block.* Paths with IDs 1 and 2 start from basic block A. Based on their respective frequencies, the probability of executing 1 to completion is 0.8. Similarly, the probability of executing 2 to completion is 0.2. Larger numbers ($max = 1$) indicate amenable paths for specialization since they exit less often.

Discussion : Overall, we observe that paths within workloads have varied characteristics; i.e path outlines are clearly visible in Figure 6. In a few cases there is overall similarity amongst a subset of paths. Examples of such cases include *.mcf, 453.soplex, sar-*, and swaptions.

Fig. 6. **D1** : Total Ins, **D2** : Guards, **D3** : ϕ's Simplified, **D4** : Total Live Vals, **D5** : Path Predictability **D6** : Path Coverage



Along dimension **D1** (size) some workloads are linearly spaced out. Examples of such workloads are 183.equake, 186.crafty, blackscholes and freqmine. Workloads such as bodytrack and streamcluster have little to no variability in the size of the longest path. This is frequently observed where paths are spatially colocated.

D2 enumerates the number of guards introduced by eliminating conditional branches. Workloads such as 179.art, 456.hmmer, fft-2d and bodytrack show significant overlap along this axis. 450.soplex has 4 paths with the same number of guards (4). The same holds for four paths in streamcluster (paths #1-#4). It is interesting to contrast **D2** (guards) with **D1** (size). For example in 183.equake, four paths are linearly spaced out across **D1** and **D2**. 186.crafty has 3 paths of similar size but differing number of guards. In 179.art, paths #3 and #5 have similar number of guards but differ in size. One cause where such behaviour is observed is due to unbalanced if conditions.

In most workloads (21 of 29), the path with the largest number of instructions is also the path with the largest number of guard checks. This does not hold true for 186.crafty, 444.namd, 464.h264ref, blackscholes, ferret, fft-2d, freqmine and sar-backprojection.

Dimension **D3** enumerates the number of ϕ's simplified. 183.crafty and 197.parser have a similar number of ϕs removed while differing along axes **D2** and **D1**. dwt53 has the same number of ϕs simplified but differing number of branches (converted to guards) in each path. fluidanimate along **D1**, **D2** and **D3** have interesting characteristics. Path #4 has large size but significantly fewer guards and ϕ's simplified.

The sum of live input and output values is shown along dimension **D4**. It is interesting to compare the value along **D4** with **D1**, i.e whether the number of live in and live outs is proportional to the size of the path. For 21 of the 29 workloads the largest path also has the largest number of live values. The converse is true for streamcluster, fluidanimate, 450.soplex, ferret, bodytrack, fft-2d, 470.lbm and 464.h264ref. For each workload there is significant variability across paths with respect to live values. There are a few workloads where 3 or more paths have similar live values. Some examples are ferret, bodytrack, 444.namd and 164.gzip. In many workloads, accounting for the largest number of live values per path will waste 25% or more of the local scratchpad. This holds in 9 of 29 applications; one path has 25% more live values than others.

Path predictability **D5** is the measure of probability a path will execute to the end. Values close to one are desirable as they imply lower overheads for specialization. Overheads are incurred when partial execution on a specialized unit needs to be rolled back. The path also needs to be evaluated in software, restarting at the beginning. Five of the 29 workloads, 470.hmmer, 444.namd, 456.hmmer, fft-2d and freqmine, have frequent paths that are perfectly predictable for the given input data. In 17 out of 29 workloads, the largest path had near perfect predictability. Conversely, 8 out of 29 workloads had large paths with poor predictability ($< 50\%$).

Path coverage is shown along **D6**. It is representative of the amount of work each path does. It is computed as the frequency weighted size of each path. Often the largest path (**D1**) does not have the highest coverage. Some examples are 164.gzip, 401.bzip2, blackscholes and fluidanimate.

Overall, there are interesting paths that stand out across workloads. Path #1 from freqmine is the largest path and has the highest number of live values and coverage, yet it is perfectly predictable for the given input. 444.namd has a few paths oriented along the **D2**-**D5**, i.e paths that have many guards yet are predictable. For some paths in bodytrack, streamcluster and 458.sjeng, increased ϕ simplification was observed along more predictable paths. Path #1 from 470.lbm has the maximum values along 5 of the 6 axes. While being the largest path, with the highest number of guards and ϕ simplifications and high coverage, it has fewer live values. Path #5 in 179.art has the maximum along all axes apart from **D5**, i.e it is a large (174 instructions) path with perfect predictability, few ϕ's simplified (3) and 6 live values. These characteristics make the path amenable for specialization. On analysis of the source for the particular path, we find lines 140–146 in `scanner.c`. This is a segment from the function `simtest2`. The code for the path is shown in Listing 2 (reformatted for typesetting).

Listing 2. Path from simtest2 – 179.art

```
Su = ((double)numf1s*su2-su*su)/
  ((double)numf1s*((double)numf1s-1.0));
Su = sqrt(Su);
Sp = ((double)numf1s*sp2-sp*sp)/
  ((double)numf1s*((double)numf1s-1.0));
Sp = sqrt(Sp);
numerator = (double) numf1s * sup - su * sp;
denom = sqrt((double) numf1s*su2 - su*su) *
  sqrt((double) numf1s*sp2 - sp*sp);
r = (numerator+e)/(denom+e);
```

Changes in 181.mcf to 429.mcf in SPEC2000 to SPEC2006 are described [25] as *''Because there have been no significant errors or changes during the years 2000 - 2004, most of the source code of the CPU2000 benchmark 181.mcf was not changed in the transition to CPU2006 benchmark 429.mcf. However, several central type definitions were changed for the CPU2006 version by the author"*. For mcf, overall path characteristics remain the same between versions. Path #5 has increased coverage. On analysis of the source, we find a new condition that changes the memory reallocation criteria, which in turn make paths in 429.mcf more amenable to specialization (increased coverage). The condition is `if( net->n_trips <= MAX_NB_TRIPS_FOR_SMALL_NET )` in `implicit.c`.

470.lbm has a total of five dynamically executed paths. Of these three are represented on the chart as area filled polygons. The other two consist of a single basic block and have zeros for dimensions **D2**, **D3** and **D4**, i.e they have no guards, ϕ's simplified (artifact of being a single block path) and no live values. They are perfectly predictable as there is only one basic block. They are not shown on the radar chart.

To summarize, we find that distinct paths across workloads have characteristics that are unique to the paths themselves.

Analysis of characteristics at the function or coarser granularity blends the characteristics from many paths and adds noise. For the purpose of specialization, we advocate the adoption of a path granularity analysis.

V. Path Derived Workload Suite

The previous sections have established the significance of characterization at a path granularity and discussed at length the characterization of a large number of workloads. The final contribution of this work is the creation of a derived benchmark suite to assist the computer architecture research community. We have identified frequent acyclic paths across 29 workloads drawn from popular benchmark suites. Using our LLVM tool chain we have outlined the paths into independent functions free from control flow (apart from guard checks). These can now be easily analyzed using existing tools for dynamic analysis such as Intel Pin. We will make available binaries of the workloads with outlined frequent paths to interested researchers in accordance with respective licensing terms.

A. Memory Address Entropy Analysis

In this section we describe how our derived suite assists researchers perform precise analyses using existing tools. To compute the memory address entropy along a path, we extended an existing memory address tracing tool. A flag was added to enable / disable trace dump at runtime. We then added instrumentation to set and unset the flag at function invocation and return via the `IMG_AddInstrumentFunction(...)` interface. The code that targets the path is shown in Listing 3.

Listing 3. Pintool Modifications

```
string name =  PIN_UndecorateSymbolName(
    RTN_Name(rtn), UNDECORATION_NAME_ONLY);
if (name.find(string("__offload_func")) !=
    string::npos) {
    RTN_Open(rtn);
    RTN_InsertCall(rtn, IPOINT_BEFORE, (
        AFUNPTR) EnterROI, IARG_END);
    RTN_InsertCall(rtn, IPOINT_AFTER, (
        AFUNPTR) ExitROI, IARG_END);
    RTN_Close(rtn);
}
```

All the outlined paths have the following naming convention `__offload_func_XXX` where `XXX` is the identifier of the path computed by the Ball-Larus algorithm [1] (see § II for more details). When control flow reaches the starting basic block of the outlined path, it optimistically chooses to invoke the *"path outlined as function"*. Providing a clean abstraction for the path allows us to write tools to target that particular region only. Should control flow deviate from the path, the function returns false to indicate a side exit, and the original program code is executed after the original program state is restored. The details of the implementation are beyond the scope of this work.

VI. Related Work

a) Benchmarks and Synthetic Workloads: Several approaches have been proposed to construct synthetic benchmarks that are representative of the real workload for specific microarchitectural behavior (e.g., cache misses [17] or branch predictability [15]). These techniques typically measure the microarchitectural runtime behavior of the real workload and construct a set of synthetic code regions that place similar demand on hardware resources. It is not sufficient for an acceleratable region to simply demonstrate statistically similar behavior to be useful for computer architects. The benchmark must be functionally representative since accelerators by definition are functionally specialized for the targeted code. Prior work has largely focused on benchmarks that mimic a real workload's power consumption [14] memory locality behavior [24], cache hit/miss ratios or even branch behavior. Bell et al. [3] presented a framework for the automatically synthesizing benchmarks from executables. They leveraged statistical simulation theory and generate C-code and assembly instructions that accurately model the workload attributes. Performance cloning [15] is another technique that seeks to more precisely capture the control flow and memory locality predictability of the original application. In contrast, our goal is to demarcate program regions in existing applications to indicate explicitly to simulators and binary analysis tools the code paths that are suitable for hardware acceleration. The demarcated paths within the original program precisely capture the functional behavior of the dynamic execution of the code paths.

b) Accelerator Studies: : Current accelerator studies have developed compiler infrastructure for studying existing CPU workloads [12], [13], [19]. They identify specific code behaviors within the workload targeted by their hardware and have provided mechanisms to demarcate them in the application. Unfortunately, the compiler approaches are closely tied in with the hardware accelerator, and it is unclear whether the identified code regions can be used by researchers developing a different accelerator architecture. A key challenge with current accelerator studies is that it is not feasible to compare accelerator architectures directly since they may not even be commenting on the same code region. Our focus has been to approach the question of "acceleratability" from the application's perspective and demarcate code paths for which specialization can directly provide performance and power benefits to the application. This permits different accelerator microarchitectures to be directly comparable since they target a common code region. Machsuite [20] provided a set of kernels drawn from various algorithms. It is unclear yet whether accelerators should be targeted at fine-granularity regions such as kernels and also whether kernels can be representative of real programs that include frequent control and are not necessarily written as a collection of kernel workloads. Our goal has been to identify the code paths within existing CPU workloads that accelerators should target.

VII. Conclusion

We advocate analysis at the granularity of acyclic program paths when assessing the amenability for acceleration of a workload. This is a new facet to the existing dynamic and static approaches. We have shown how this can be done in a scalable manner via lightweight dynamic instrumentation and static reconstruction. We have built a robust LLVM based toolchain to automate the analysis and presented our results for 29 workloads drawn from SPEC2000, SPEC2006, PERFECT and PARSEC. We have analysed $\simeq 356K$ paths across workloads and presented data for 143 paths. Our results show that within a workload, paths have disparate characteristics. Summarized characteristics at coarser granularities blend these characteristics and may draw imprecise conclusions.

References

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

[2] K. Barker, T. Benson, D. Campbell, D. Ediger, R. Gioiosa, A. Hoisie, D. Kerbyson, J. Manzano, A. Marquez, L. Song, N. Tallent, and A. Tumeo. *PERFECT (Power Efficiency Revolution For Embedded Computing Technologies) Benchmark Suite Manual*. Pacific Northwest National Laboratory and Georgia Tech Research Institute, December 2013. http://hpc.pnnl.gov/projects/PERFECT/.

[3] R. H. Bell and L. K. John. Efficient power analysis using synthetic testcases. In *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC'05)*, pages 110–118, Oct. 2005.

[4] 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*, pages 1–12, 2012.

[5] C. Bienia. *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University, January 2011.

[6] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC benchmark suite: characterization and architectural implications. In *PROC of the 17th PACT*, 2008.

[7] 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 *Proceedings of the 19th ACM/SIGDA international symposium on Field programmable gate arrays*, pages 33–36. ACM, 2011.

[8] 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.

[9] L. Eeckhout, R. H. Bell, B. Stougie, K. De Bosschere, and L. K. John. Control flow modeling in statistical simulation for accurate and efficient processor design studies. In *Computer Architecture, 2004. Proceedings. 31st Annual International Symposium on*, pages 350–361. IEEE, 2004.

[10] M. Ferdman, A. Adileh, O. Kocberber, S. Volos, M. Alisafaee, D. Jevdjic, C. Kaynak, A. D. Popescu, A. Ailamaki, and B. Falsafi. Clearing the clouds: a study of emerging scale-out workloads on modern hardware. In *Proc. of the seventeenth ASPLOS*, 2012.

[11] Google. Google performance tools.

[12] 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.

[13] 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.

[14] C.-T. Hsieh and M. Pedram. Microprocessor power estimation using profile-driven program synthesis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 1998.

[15] A. Joshi, L. Eeckhout, R. H. Bell, and L. John. *Performance Cloning: A Technique for Disseminating Proprietary Applications as Benchmarks*. Oct. 2006.

[16] C. Luk, R. S. Cohn, R. Muth, H. Patil, A. Klauser, P. G. Lowney, S. Wallace, V. J. Reddi, and K. M. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *PLDI*, 2005.

[17] J. Meng, X. Wu, V. Morozov, V. Vishwanath, K. Kumaran, and V. Taylor. SKOPE. In *Proceedings of the 11th ACM Conference on Computing Frontiers*, 2014.

[18] T. Nowatzki, V. Gangadhar, and K. Sankaralingam. Exploring the potential of heterogeneous von neumann/dataflow execution models. IEEE Computer Architecture Letters, 2015.

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

[20] 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.

[21] Y. S. Shao and D. Brooks. ISA-independent workload characterization and its implications for specialized architectures. In *2013 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 245–255. IEEE, Apr. 2013.

[22] 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.

[23] Y. S. Shao, S. Xi, V. Srinivasan, G.-Y. Wei, and D. Brooks. Co-designing accelerators and soc interfaces using gem5-aladdin. In *Microarchitecture, 2016. MICRO-49 2016. 49th International Symposium on*, 2016.

[24] E. S. Sorenson and J. K. Flanagan. Evaluating synthetic trace models using locality surfaces. In *Workload Characterization, 2002. WWC-5. 2002 IEEE International Workshop on*, pages 23–33, Nov 2002.

[25] SPEC. 429.mcf - SPEC CPU2006 Benchmark Description.

[26] L. Wu and M. A. Kim. Acceleration targets: A study of popular benchmark suites. In *The First Dark Silicon Workshop, DaSi*, 2012.

[27] L. Yen, S. C. Draper, and M. D. Hill. Notary: Hardware techniques to enhance signatures. In *Microarchitecture, 2008. MICRO-41. 2008 41st IEEE/ACM International Symposium on*, pages 234–245. IEEE, 2008.