2023 International Symposium on Computer Architecture and High Performance Computing Workshops (SBAC-PADW)

Quinn Pham
University of Alberta
Edmonton, AB, Canada
{gobran, qpham}@ualberta.ca

José Nelson Amaral
University of Alberta
Edmonton, AB, Canada
{joao.carvalho, jamaral}@ualberta.ca

Nemanja Ivanovic
IBM Corporation
Markham, ON, Canada
{kbarton, nemanjai}@ca.ibm.com

***Abstract*—A new microprocessor within a given processor architecture may introduce performance-improving features that either can only be accessed through novel instructions or require new code-generation techniques to be beneficial. In response, compilers must be extended/improved to make use of these new instructions and to generate better schedules for the new hardware. The compiler improvements that enable these specializations can take significant time to develop, thus applications compiled Ahead-Of-Time (AOT) will often not benefit from code specialization without later recompilation. Furthermore, code compiled for a specific hardware sub-target lacks performance portability, thus, for better performance, there is a need to maintain multiple builds for each processor architecture leading to significant development and maintenance costs. This paper demonstrates that such challenges can be overcome by applying code specialization only to a small percentage of the code in a program. Moreover, it proposes DASS, a novel Dynamic Adaptive Sub-Target Specialization technique to recompile selected parts of a program at runtime. Empirical evidence indicates that selective specialization can achieve up to 93% of whole-program specialization speedup by statically specializing less than 1.5% of the application code. Furthermore, DASS can dynamically achieve performance close to that of static specialization, reaching up to 83% of statically attainable speedup while performing recompilation and redirection during execution.**

I. Introduction

Performance-improving features introduced as a processor architecture evolves over time, either can only be accessed through new hardware instructions or require changes in code generation to yield better performance [1]–[6]. For instance, most modern CPUs and GPUs have specific instructions to exploit data parallelism following the **Single-Instruction Multiple-Data** (SIMD) paradigm. SIMD computations are executed in modern hardware via vector instructions that operate on wide vector registers (*e.g.* 512 or 4096 bits) [7], [8]. Acceleration through specialization is not new but it is a growing trend [9]–[11]. For instance, major companies have introduced specialized units into their commodity processors, namely **Advanced Matrix eXtensions** (AMX™) by Intel®, **Scalable Matrix Extensions** (SME™) by Arm®, and **Matrix Multiply-Assist** (MMA™) by IBM®. These units require new instructions that programmers/compilers need to explicitly include in their code to benefit from the full potential of these extensions.

Most performance-critical applications are distributed as **Ahead-Of-Time** (AOT) compiled binaries by **Independent Software Vendors** (ISVs) [12], [13]. An alternative to AOT compilation is **Just-In-Time** (JIT) compilation that delays optimization and code generation until application runtime (Section II-C). Application binaries compiled AOT can execute immediately and an indefinite number of times, without any compilation costs at runtime. Moreover, hardware architectures are designed to be backward-compatible, meaning that all instructions available on older architecture versions must execute on newer architecture releases. Thus, AOT-compiled binaries that use a common subset of **Instruction Set Architecture** (ISA) instructions can execute without recompilation on different releases of the same architecture. Nevertheless, AOT-compiled binaries cannot benefit from the performance of specialized computing units in most commodity processors without recompilation — the new instructions need to be added/generated by programmers/compilers. Furthermore, compilers can be modified to generate a schedule of existing instructions that executes faster on new versions of an architecture [14]. Thus, recompiling portions of AOT-compiled applications with the latest version of a compiler may result in binaries produced with a better code-generation strategy.

This paper presents results that indicate that the benefits of novel hardware instructions and/or better code-generation strategies can be achieved by specializing only parts of an application — henceforth referred to as code segments or simply segments. In this work, code segments are selected, based on their frequency of execution and their contribution to the run time of the application, for specialization either via code attributes — without any changes to existing AOT compilers — or dynamically — through a novel JIT-enabled technique. Experimental results with SPEC CPU® 2017 benchmarks [15] indicate that dynamic specialization can enable ISVs to deliver close to the fully-specialized performance. This paper makes the following contributions:

- An in-depth performance evaluation of the SPEC CPU 2017 benchmarks, which shows that specializing only a small fraction of the application code closely matches the performance of a fully specialized program. For instance, specializing only 1.5% of the *Imagick* benchmark leads to 93% of the performance attained by whole-program specialization. For the *LBM* benchmark, whole-program specialization can be matched with specialization of only 19% of the application.
- Dynamic Adaptive Sub-Target Specialization (DASS), a compiler system that enables segments of an AOT-compiled program to be recompiled at runtime to take advantage of sub-target specialization and relevant improvements to the compiler since the program was compiled. A proof-of-concept DASS implementation in LLVM [16] enables close to AOT-specialized performance for some benchmarks from the SPEC CPU 2017 suite.

The remainder of this paper is organized as follows. Section II presents background concepts and terminology adopted throughout the paper. Section III presents the main ideas behind DASS and a description of the proof-of-concept implementation in LLVM. Section IV describes the experimental setup and methodology used to obtain preliminary results, and identifies instances where DASS bridges the limitations of AOT compilation w.r.t. new hardware and compiler versions, and where it falls short. Section V contrasts related works with DASS. The conclusions of this study are discussed in Section VI.

II. Contrasting SS with AOT and JIT Compilation

This paper focuses on optimization benefits enabled by **Sub-Target Specialization** (Section II-A), through which a program is optimized

979-8-3503-8160-3/23/$31.00 ©2023 IEEE
DOI 10.1109/SBAC-PADW60351.2023.00016

for a specific model of a CPU architecture (microarchitecture). Sub-target specialization is a performance optimization frequently missed in common software due to how it reduces program portability. **Independent Software Vendors** (ISVs) leverage program portability to decrease maintenance and distribution costs of projects with large code bases (e.g. millions of lines of code). For performance-critical software, ISVs utilize **Ahead-Of-Time** (AOT) compiled (Section II-B) languages — *e.g.* C/C++, Fortran, and COBOL — high-level languages that need to be compiled to the assembly of a given architecture before execution. In comparison, **Just-In-Time** (JIT) compiled (Section II-C) languages — *e.g.* Java, Python, and C# — can start execution immediately, adapting to the currently executing microarchitecture, and having some or all of the program (re)compiled at run-time.

A. Sub-Target Specialization (SS)

Most optimizing compilers, by default, generate code that is generic to a given CPU architecture, or **target**, but specialization for a given ISA version, or **sub-target**, can be enabled via compiler flags or language attributes. CPU vendors develop new ISAs over successive hardware releases as the hardware design changes to meet the needs of modern software development [6]. Code that is not specialized to a sub-target can execute on any CPU of the same target, as CPU vendors maintain backward compatibility with previous releases by retaining, or in some instances emulating, features from previous versions of the ISA [17], [18]. However, this portability comes at the cost of code with no sub-target specialization that might not take full advantage of the new ISA or improvements in the compiler. ISVs usually avoid sub-target specialization to decrease the costs of developing, maintaining, and distributing software. Avoiding sub-target specialization creates a gap between the potential performance attainable by a program on a given hardware, and the actual performance achieved by a generic build of the same program. For processor designers and manufacturers, this gap is worrying because it reduces the performance advantage of upgrading to new hardware.

The benefits of specialization are further influenced by the compiler used to build a program. Some compilers may support more complex specialization for a given sub-target (often the case with compiler vendors that develop CPU architectures). Thus, the aforementioned performance gap between specialized and generic code depends heavily on the available compiler.

B. Ahead-Of-Time (AOT) Compilation

AOT compilation is the process of fully translating a program to an executable binary format, prior to the execution of the program. Many languages (`C/C++`, `FORTRAN`, `COBOL`, etc.) [19]–[22] were designed for, and are mainly used through, AOT compilation. AOT compilation enables the use of these languages for high-performance computing because the full compilation allows for extensive and complex optimizations to be performed without impacting execution time. However, with optimization, the target hardware for the program must be known ahead of time. The target architecture for code generation can be set implicitly — the program is generated for the hardware where the compilation is executed — or explicitly by specifying either a generic target architecture or a specific sub-target.

The cost of AOT compilation is incurred before execution and depends on the complexity of the source language as well as the size of the source code [23]–[25]. There is zero execution-time overhead, for compilation, because the entire program, including dependent libraries, is assembled beforehand. Thus, AOT compilation is advantageous for programs with many Lines of Code (LOC) because the high compilation cost is separated from program execution. The AOT cost is incurred only once and can be amortized over many program executions. Thus, AOT compilation is appealing to performance-focused ISVs whose clients need programs that run efficiently.

C. Just-In-Time (JIT) Compilation

JIT compilation consists of translating a part of a program (functions or code blocks) into executable binary code during execution time [26]. Instead of the program being compiled before execution, code is typically interpreted before specific sections are identified as candidates for JITing. Delaying JIT compilation can speedup startup times for the program, while only paying the AOT compilation cost for small parts of the program [27]. Furthermore, the JIT process can be offloaded to a separate thread of the program, allowing for parallel execution and compilation. However, the compilation time can still impact the overall execution time of the program, thus, identifying suitable JIT candidates is key. In traditional JIT systems, this is guided by mechanisms that identify frequently executed code (hot code) which is then marked and compiled as the program executes [28]. Ideally, the compilation overhead is offset by the speedup on future executions, which payout if the JIT-compiled code re-executes often.

III. Dynamic Sub-Target Specialization

The central idea of this paper is to introduce dynamic adaptive sub-target specialization to AOT-compiled applications that could run on processors with a common target but different sub-targets. At runtime, DASS performs sub-target specialization on code segments copied in a compiler intermediate representation (IR) form at compile time. The IR of such code segments is stored alongside the code in the program binary, similar to other fat binary approaches [29]. Aside from sub-target specialization, DASS also enables applications to benefit from improvements in new releases of a compiler without AOT recompilation. The remainder of this section presents DASS's core design (Section III-A) — from code segment cloning to dispatch and execution of sub-target-specialized code at runtime — and a proof-of-concept implementation in LLVM (Section III-B) used to evaluate DASS in Section IV.

A. Core Design

DASS is a compiler system that compiles the application code and produces an AOT-compiled binary that is optimized for a specific target, as most optimizing compilers. However, DASS binaries are augmented with the IR of selected code segments that could be further specialized for a sub-target at runtime. Candidate segments may be identified through profiling information, static analysis, and/or run-time cost/benefit heuristics. The main steps to produce a DASS binary that is enabled to benefit from sub-target specializations are: 1) *Fat Binary Creation*: involves cloning and storing the IR of selected segments alongside the generated code (Section III-A1); 2) *Dynamic Redirection Setup*: generation of instructions to select, at runtime, if each selected segment will execute the AOT-compiled code with target optimizations or the JITed[1] code with sub-target optimizations (Section III-A2); 3) *Symbol Resolution*: this step enables the JITed code to address global variables and call functions in the AOT-compiled code or in libraries (Section III-A3); 4) *Dynamic Sub-target Specialization*: generation of sub-target-specialized code at run-time (Section III-A4). Both *Fat Binary Creation* and *Dynamic redirection setup* happen at compile-time, while *Symbol Resolution* and *Dynamic Sub-Target Specialization* happen at run-time.

[1]JITed is a neologism for just-in-time compiled.

1) Fat Binary Creation: Alongside the AOT-compiled application code, DASS stores the following for each selected code segment: 1) `SEG.ID`: a unique identifier used by DASS to know which segment to use and/or compile; 2) `SEG.IR`: the code segment's IR; and 3) `SEG.SYM_LIST`: the list of symbols used in the IR (*e.g.* variables and/or functions). `SEG.IR` is copied after common simplification compiler transformations — *e.g.* dead-code elimination and control-flow graph simplification — to reduce the IR prior to cloning but prior to any sub-target transformation passes. Variables that are constant and statically initialized are not added to `SEG.SYM_LIST` in order to not obfuscate statically known constants and allow common optimizations — *e.g.* constant folding or constant propagation — to happen during dynamic specialization (Section III-A4).

2) Dynamic Code Redirection: After IR cloning, DASS generates code that dynamically redirects execution to the AOT-compiled code while the JITed code is not available. The JITed segment code might not be available because `SEG.IR` is being compiled. `SEG.IR` compilation can happen at any point during the execution of the program. For example, the first time the execution reaches the entry of the segment or the beginning of the program execution. JITed code might also not be available because, based on a run-time cost/benefit analysis, DASS decided to use the AOT-compiled code instead. Once the JITed code is available, the redirection code inserted by DASS can direct execution to use it.

3) Symbol Resolution: Any access to symbols in the AOT-compiled code needs to be resolved to their addresses because `SEG.IR` is compiled at runtime. `SEG.SYM_LIST` contains symbols used by each code segment and, after the link stage, also contains the address to each symbol. DASS resolves each symbol by associating the symbol with their respective address in `SEG.SYM_LIST` after compiling `SEG.IR`. This resolution ensures that the dynamic linker knows the address of all symbols referenced by the JITed code. Similarly, DASS resolves any function calls from JITed code to functions in the AOT-compiled code to their addresses through the `SEG.SYM_LIST`. Calls from the JITed code to library functions are handled by the dynamic linker, as is commonly done for any AOT-compiled program.

4) Dynamic Sub-Target Specialization: DASS performs sub-target specialization by setting the target and sub-target for compilation to the detected host CPU executing the application at runtime. `SEG.IR` compilation overhead can be amortized by spawning the compilation on a worker thread to overlap it with the program execution.

B. Proof-Of-Concept Implementation in LLVM

The **proof-of-concept** (POC) implementation of DASS is realized inside the LLVM project [16] with whole functions as the granularity for sub-target specialization. LLVM is an umbrella project that hosts sub-projects with tools, libraries, and infrastructure for, among many other things, creating compilers, writing transformation passes, and building JIT compilers. LLVM provides a production-grade Intermediate Representation called **LLVM IR** used for target-independent code analysis and transformation.

This section describes 1) how DASS extends Clang for explicitly marking functions to be JITed (Section III-B1); 2) an LLVM IR pass that copies the IR and inserts dynamic-dispatch logic into marked functions (Section III-B2); 3) a JIT runtime library built on top of LLVM's ORC JIT (Section III-B3); and 4) extensions to LLVM's linker (LLD) to create sections in the binary that store the IR of JIT candidate functions and other DASS metadata (Section III-B4).

```
int foo(int x) __attribute__((ijit));
```

Listing 1: Function annotated with the ijit attribute.

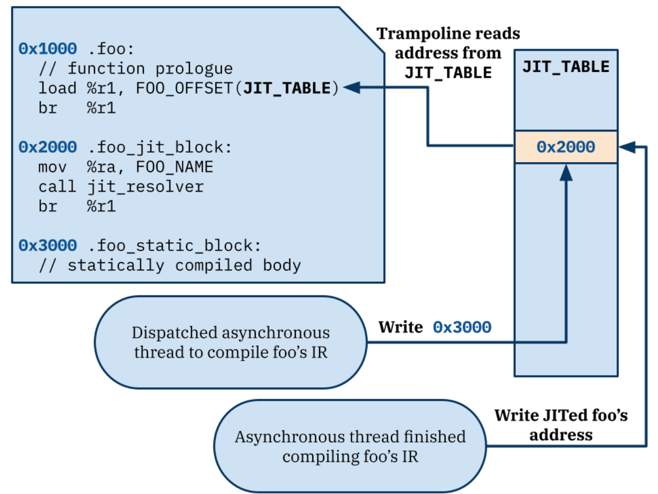

Fig. 1: Diagram of the JIT call trampoline.

1) Extensions to Clang: In the POC implementation functions with the **`ijit`** function attribute (Listing 1) are marked for sub-target specialization through JITing. As discussed in Section III-A, candidate functions can be identified through profiling, static analysis, or run-time cost/benefit analysis. In this POC implementation, no automatic mechanism to select functions is implemented. Instead, a simple criteria based on the frequency of execution and the fraction of execution time attributed to a function guides manual application of the attributes to the functions of an application. (See Section IV-C2).

2) IR Cloning and Dynamic-Dispatch Insertion Pass: This LLVM IR pass copies function definitions that have the **`ijit`** attribute and serializes their IR into constant strings. The IR strings are added to a list of DASS metadata objects that contain: 1) the name of the copied function; 2) its IR; and 3) a list of symbol names and their addresses — the latter are filled in by the linker. These symbols are variables or functions used by the copied function and their addresses are kept so that the JIT runtime can bind them at runtime after the IR is JITed (Section III-B3). All the information obtained by DASS's IR pass is kept under compiler-reserved variables[2] that are stored in dedicated binary sections (Section III-B4).

Once the marked functions are copied, the DASS's IR pass inserts a trampoline at the entry point of the copied functions' body. The trampoline mechanism is outlined in Figure 1. On function call, the trampoline loads an address from a table and jumps to it. The table entry can point to 1) the original entry point of the function, 2) a path that calls DASS's JIT runtime to dispatch asynchronous JITing of the copied IR, or 3) a path that calls the JITed IR code. Initially, the trampoline always takes the path to call DASS's runtime. The runtime dispatches an asynchronous thread to compile the IR string, if one was not already dispatched, and atomically updates the JIT table with the address of the statically compiled function body. Once the IR string is compiled, then the table is atomically updated thus future trampoline loads will jump to the JITed code.

In the POC implementation, functions are copied right after the function simplification passes in LLVM, which is fairly early in the optimization pipeline. The continuation of this research will investigate alternative stages of the optimization pipeline in which candidate functions should be copied. Optimization passes that do not depend on sub-target-specific information should be executed before

[2]Not accessible by the application code.

cloning so that the costs do not impact application execution time. Nevertheless, the initial experimental results indicate that complete optimization pipelines can be executed at runtime with reasonable cost (Section IV). Therefore, not performing AOT compiler transformations that depend on sub-target information is just a policy to avoid premature specialization.

3) JIT Runtime: DASS's runtime is built on top of **On Request Compilation** (ORC) v2 API [30]. ORC is a library for building JIT compilers. Languages such as `Lua` [31], `C++` [32], and `Swift` [33] make extensive use of ORC's API to build their interpreters and **read-eval-print loop** (REPL) tools. DASS employs ORC's methods to explicitly define symbol bindings — used to handle references from JITed code to functions and variables in AOT-compiled code or libraries — and compile IR strings upon request.

The JIT runtime's entry function is called through the trampoline inserted by DASS's IR pass (Section III-B2). As arguments, the entry function takes the name of the function to JIT compile, the trampoline table entry associated with said function, the address to the entry of the AOT compiled function, and the address to the block that calls the JITed function via the trampoline. An asynchronous thread is created to JIT the requested function and control is immediately returned to the application code. While the JIT compiles the candidate function, the AOT compiled code is executed. Once the JIT compilations are complete, the trampoline table is atomically updated to direct the application to the block that calls the JITed function's code on subsequent trampoline table loads.

ORC is built to allow JITed code to execute in multiple threads or to spawn new threads. JITing concurrency is also available out of the box. Thus, the only additional synchronization required by DASS is to ensure that only one worker thread is compiling a particular function at a time. This is guaranteed by an atomic update to the trampoline entry used to indicate that a function 1) needs to be compiled; 2) is being compiled by another worker thread; or 3) was already compiled. In the two latter cases, the JIT runtime immediately returns the control flow to the application code because the trampoline table is/will be updated with the JITed code address by the worker thread JITing said function. Any worker thread that successfully exchanges the table entry from 0 to 1 acquires the task to JIT the function's IR. The value of 1 in the table entry causes the trampoline to direct control flow to the AOT-compiled code.

DASS sets the sub-target for JITing with the host CPU information detected by ORC, which is able to identify the CPU running the application. In the current POC, every function is compiled with the same optimization pipeline. However, the implementation is flexible and easily adaptable to enable functions to be optimized with different sets of passes.

4) Extensions to LLD: DASS requires very minimal changes to LLVM's linker, LLD. More specifically, LLD was extended to concatenate the list of DASS metadata objects generated for each compilation unit by DASS's IR pass (Section III-B2) into a single global hidden list. Additionally, the start and end symbols of DASS's JIT binary sections are generated by further extensions to LLD. Such symbols are used: 1) to skip the JIT-runtime initialization if no function was marked to be JITed; and 2) to enable the runtime to safely traverse the entire JIT section.

C. Sources of Overhead in Current Implementation of DASS

DASS's current implementation has inherent overheads due to design decisions made during its development. The main sources of overhead, observed in the experimental evaluation of DASS (Section IV), are listed and discussed below.

1) Compilation Overhead: Because DASS moves sub-target specialization from compilation time to run time, an obvious source of overhead is the cost of compiling the IR of selected segments. Compilation cost is a non-trivial function of the size — measured as the number of instructions – and features — *e.g.* presence of loops, number of accesses to memory through global symbols or via pointer arguments — of the copied IR. The size can determine the workload of compiler transformation passes while features determine which passes may execute and how often some passes may execute during compilation. Moreover, different optimization pipelines (*e.g.* `-O2` and `-O3`) have different costs as they enable different passes, or configure the same passes with different assumptions on the desired trade-off between expected compilation time and expected performance of the generated code. Even if the compilation is fully overlapped with the execution of the application, longer compilation times delay the use of the potentially more optimized JITed code and thus indirectly add overhead. For the applications used in this paper's experimental evaluation (Section IV-E), compilation time was not a significant source of overhead — up to 0.46% of an application's execution time (See Table III).

2) Initialization Overhead: This overhead comes from setting up the JIT runtime as well as ORC and LLVM's components for JITing. The setup entails registering each selected segment's IR and their corresponding symbols, setting the sub-target, and selecting the optimization pipeline. Initialization overhead cannot be easily amortized because the IR compilation cannot happen before the JIT runtime is fully started. However, as discussed in Section IV, initialization cost is usually minimal — up to 0.006% of the application execution time, with small variations based on the number of symbols and the size of a segment's IR.

3) Indirection Overhead: This overhead comes from the trampoline structure used as a dynamic-dispatch mechanism (Section III-A2). The cost of executing the table load and indirect branch is incurred every time a selected function is called. For frequently called functions, this indirection overhead can become significant (Section IV).

A less obvious source of indirection overhead is paid when accessing variables or calling functions in the AOT-compiled binary. The address of variables and functions defined in the AOT-compiled binary is stored in a table, thus every access goes through a double indirection. Some applications exhibited a high indirection overhead due to extra memory operations to access data and calling functions in the AOT-compiled binary (Section IV). This overhead can be eliminated by using instructions that directly reference the address of such symbols, which might not be possible with the code model used by the JIT compiler.

4) Lower Code & Data Locality Overhead: This overhead comes from the decreased data and instruction locality between the JITed code and the AOT-compiled code. The JITed code is generated into a shared library that is loaded at run-time by the dynamic linker. Thus, if the JITed function is placed far away — *e.g.* different memory pages — from the AOT-compiled code that calls it, then performance suffers from instructions cache and iTLB misses. Similarly, as the JITed code is no longer co-located with the AOT-compiled code, access to variables in the AOT-compiled code might exhibit poor data-cache locality. This source of overhead can be more significant if transitions between JITed and AOT-compiled code are frequent (Section IV-E2).

IV. Is DASS a Worthwhile Approach to SS?

The proof-of-concept implementation of DASS is used to evaluate the feasibility of using dynamic sub-target specialization for `C/C++`

code through a fat-binary JIT approach. In the following sections, the evaluation aims to answer the following questions:
① How the performance of sub-target-specialized code compares to target-specialized code? (Section IV-C1)
② Can the improvements of sub-target specialization be achieved without recompiling the whole application? (Section IV-C2)
③ Can DASS achieve the improvements of static sub-target specialization? (Section IV-D)
④ What are the significant sources of overhead in the current implementation of DASS? (Section IV-E)

A. Benchmarks, Hardware, Compiler, and DASS Variations

Our evaluation is performed on the `C/C++` benchmarks from the SPEC CPU 2017 suite. Benchmarks are compiled with the SPEC *base*[3] metric and run on the `ref` workload. The default `-O3` and `-Ofast` pipelines are used to compile the `intrate` and `fprate` benchmarks respectively. All benchmarks are compiled with a development compiler built from LLVM 15.0.0, which includes optimizations for the PowerPC architecture. Experiments are run on a `POWER10` machine with 24 processor cores, and 895 GB of RAM.

Three variations of DASS are evaluated: 1) **D-Block**: selected functions are compiled on the first call and execution is blocked until the compilation completes, at which point the application execution resumes by calling the JITed code; 2) **D-Async**: selected functions are also compiled on the first call but asynchronously on a separate thread, thus the application execution continues executing the AOT-compiled function. Once compilation completes, subsequent calls use the JITed code; and 3) **D-Start**: all selected functions are compiled at once before the `main` function is called. In this variant, compilation also blocks the execution of application code.

B. Experimental Methodology

The experimental results presented are the average of three executions of a program (following the SPEC reportable runs recommended minimum [34]). The variance between executions is very small, at most 0.265% of execution time. The baseline *P7* for most speedup calculations is a benchmark specialized to `POWER7`. Performance metrics are collected through hardware counters with the Linux Profiler, `perf`. This section evaluates the performance of each benchmark when sub-target specialization is applied 1) to the *whole program*; and 2) to *selected functions*. Whole-program specialization is performed via the `-mcpu=<sub-target>`, where `<sub-target>` is set to `pwr7` (`POWER7`), `pwr8` (`POWER8`), `pwr9` (`POWER9`), and `pwr10` (`POWER10`). Sub-target specialization of selected functions is performed via the `arch=<sub-target>` function attribute and setting `-mcpu` to `pwr7`.

A function f is selected for both static and dynamic sub-target specialization based on a sub-target specialization importance criteria, measured on a `POWER10` machine: 1) the number of `perf` samples that hit f represents at least $XX\%$ of total samples in the code compiled for `POWER10`; and 2) the number of samples that hit f decreases by $YY\%$ between the profiles obtained by applying whole-program specialization for the `POWER7` and the `POWER10` sub-targets. This selection criteria is indicated as `EXX-SYY`. For example `E02-S05` selects functions with 2% of total sample count and that have 5% fewer samples when specialized for `POWER10` than for `POWER7`. The first condition establishes that f makes a contribution to the total execution time while the second establishes that specializing the compilation affects the contribution of the f to the total execution time.

[3]All benchmark modules are compiled with the same set of compiler flags passed in the same order.

C. Static Specialization Speedup

This section examines the program speedup from static specialization by comparing the execution time of benchmarks compiled for newer sub-targets, up to `POWER10`, with respect to *P7*.

1) Whole-Program Specialization: Figure 2 presents the speedup achieved with whole-program sub-target specialization. The variation from a 218% speedup in ***x264*** to a slowdown in ***mcf*** indicates that the advantage of sub-target specialization depends on the features of the program; on how the compiler generates code; and on hardware design. For instance, the significant improvement in ***x264*** comes from increased support for loop vectorization starting with `POWER8`. The addition of several vector instructions to the POWER ISA enables the compiler to vectorize the benchmark's loops, adding the `POWER9` instruction **`vabsdub`** to optimize many computations. With `POWER10` further optimization is applied to functions like **`x264_pixel_var_16x16`** where the execution time is reduced by 6× by introducing vectorization intrinsics. ***namd*** also sees substantial improvements, especially on `POWER9`, where the compiler uses instructions such as **`extswsli`** to improve execution time performance.

The use of a compiler version that is still in development where some of the profitability analyses are still being deployed explains the results where `POWER10` specialization does not produce the best speedup (*e.g.* ***mcf***, ***namd***, ***lbm***). For instance, in ***lbm*** an aggressive **`SLPVectorizer`** pass on `POWER10` introduces unnecessary vectorization. Upon removing this pass, the speedup between `POWER9` and `POWER10` becomes equal. The slowdown in ***mcf*** emerges from a 75% increase in the execution time of **`sqec_qsort`** because of a mix of less aggressive loop unrolling and changes to the **`LoopStrengthReduce`** pass.

The results in this section indicate that sub-target-specialized code achieves significantly higher speedups than target-specialized code, providing an answer to ①. However, speedup varies based on the hardware needs of the program. In applications where new hardware can be utilized (***x264***), specialization is very effective, while applications like ***xalancbmk*** show negligible speedup. These results encourage limiting the scope of optimization to code that might produce significant speedup, ignoring functions that do not benefit from specialization.

2) Selective Function Specialization: To measure the effectiveness of reducing the specialization scope to individual functions, this Section evaluates the performance attained if only selected functions, as per the selection criteria in Section IV-B, are specialized for `POWER10`. Three different criteria selections were evaluated (E10-S10, E05-S05, E02-S02), these values were chosen as they progressively select a larger portion of the benchmark code for specialization and thus allow for a better understanding of how effective more selective specialization can be. Table I shows, for each selection criteria, the percentage of IR instructions and function execution time relative to the whole program. Table II lists the functions selected for each benchmark with the E05-S05 criteria. Template functions were not considered for specialization because template specialization for each template type would need to be manually written. If selection is done in a compilation pass, then template functions pose no challenges for dynamic specialization.

Figure 3 compares the speedup from only specializing functions selected by each criteria to `POWER10`, with the non-selected functions compiled for `POWER7`, compared to whole-program specialization for `POWER10`. The effectiveness of selective specialization can be

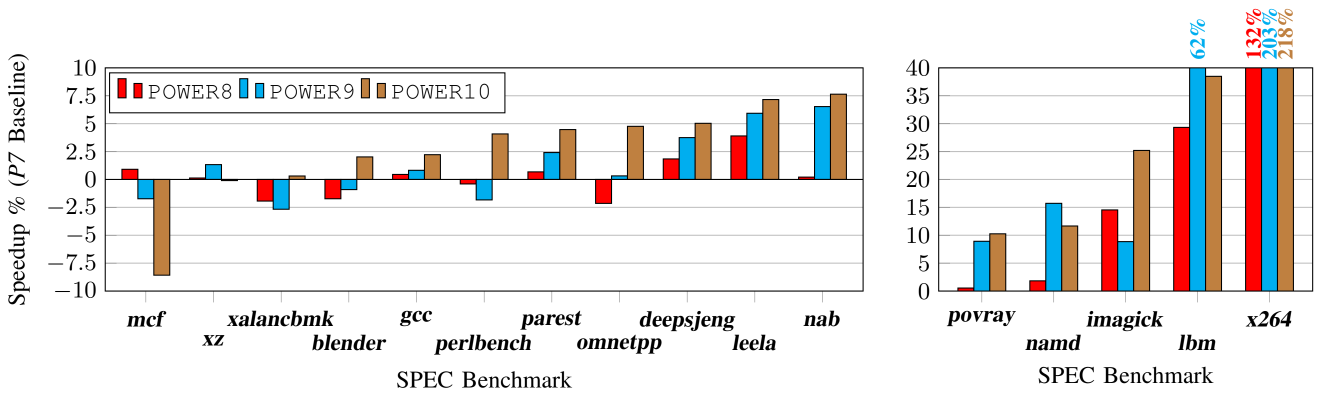

Fig. 2: Sub-target specialization speedup percentage for a given sub-target, in respect to *P7*.

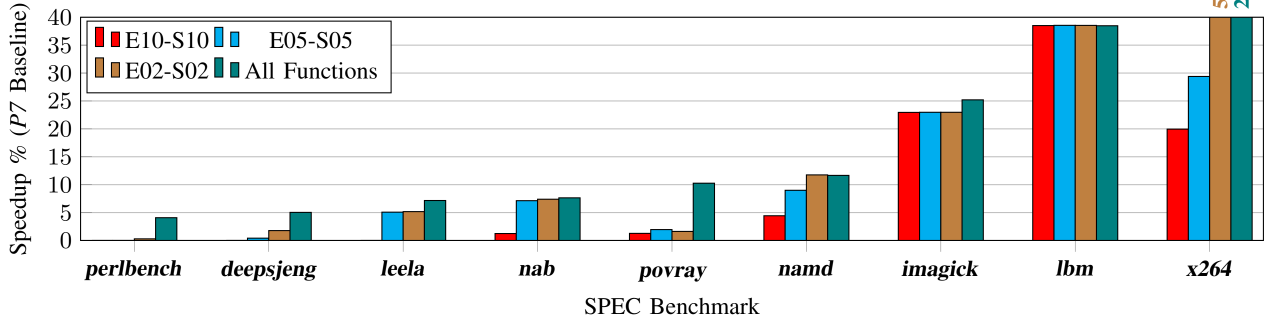

Fig. 3: Speedup percentage attained by different function specialization criteria for `POWER10` relative to *P7*.

TABLE I: Percentage of benchmark code (IR instructions) and execution time corresponding to functions selected by each criteria. "N/A" indicates that no function met a criteria. "-" indicates that the same functions were selected as the criteria to the left.

Benchmark	E10-S10		E05-S05		E02-S02	
	IR	Time	IR	Time	IR	Time
perlbench	N/A	N/A	0.26%	7.81%	0.93%	18.5%
namd	2.10%	29.0%	10.0%	72.7%	16.5%	99.5%
povray	0.09%	14.4%	0.26%	49.4%	0.81%	62.3%
lbm	19.1%	99.5%	-	-	-	-
x264	0.24%	39.6%	0.27%	46.2%	2.58%	69.1%
deepsjeng	N/A	N/A	10.1%	13.9%	25.1%	38.1%
imagick	1.48%	87.2%	-	-	-	-
leela	N/A	N/A	0.69%	21.0%	1.82%	33.8%
nab	0.29%	11.5%	4.38%	76.7%	5.32%	86.5%

better understood by comparing the execution time percentage for the selected functions in Table I with the difference between the selective specialization speedup and that achieved by specializing all functions. As an example, the results for ***nab*** are positive as the E05-S05 selection covers 76.7% of execution time, while the specialization speedup is close to equal with specializing all functions. Answering ②, reducing the scope of specialization to selected functions can realize a significant portion of specialization speedup, provided that the selected functions are hot and improve with new sub-target compilation. The function selected by the criteria in ***lbm***, **`LBM_performStreamCollideTRT`** accounts for $>$ 99% of the total samples in the profile. Thus, specializing this single function realizes almost all the improvements of whole-program sub-target specialization. In comparison, ***x264*** sees lower speedup because its specialization gains are spread across the program and some functions

TABLE II: Functions selected for the `POWER10` E05-S05 criteria.

Benchmark	Function	File	Line
perlbench	Perl_leave_scope	scope.c	759
namd	pairlist_from_pairlist	ComputeNonbondedInl.h	32
namd	calc_pair_energy_fullelect	ComputeNonbondedUtil.h	352
namd	calc_pair_energy_merge_fullelect	ComputeNonbondedUtil.h	354
namd	calc_pair_fullelect	ComputeNonbondedUtil.h	351
namd	calc_pair_merge_fullelect	ComputeNonbondedUtil.h	353
namd	calc_pair	ComputeNonbondedUtil.h	349
namd	calc_self_energy	ComputeNonbondedUtil.h	359
namd	calc_self_energy_fullelect	ComputeNonbondedUtil.h	361
povray	All_Plane_Intersections	planes.cpp	103
povray	All_Sphere_Intersections	spheres.cpp	122
povray	Inside_Plane	planes.cpp	226
povray	All_CSG_Intersect_Intersections	csg.cpp	235
lbm	LBM_performStreamCollideTRT	lbm.c	262
x264	x264_pixel_satd_8x4	common/pixel.c	234
x264	get_ref	common/mc.c	232
x264	mc_chroma	common/mc.c	263
deepsjeng	feval	neval.cpp	1043
deepsjeng	see	see.cpp	19
imagick	MeanShiftImage	magick/feature.c	2108
imagick	MorphologyApply	magick/morphology.c	3827
imagick	SetPixelCacheNexusPixels	magick/cache.c	4732
leela	kill_or_connect	FastBoard.cpp	1214
leela	is_eye	FastBoard.cpp	805
leela	get_pattern_fast_augment	FastBoard.cpp	1356
nab	mme34	eff.c	3203
nab	heapsort_pairs	nblist.c	114
nab	searchkdtree	nblist.c	667

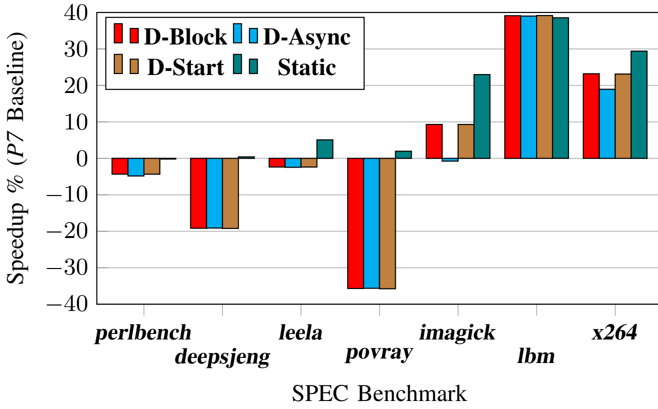

Fig. 4: `POWER10` speedup percentage for static specialization and DASS variations with the E05-S05 selection criteria.

with low execution time experience far greater speedup than the criteria selected functions. Furthermore, the relative importance of functions varies in the three workloads for ***x264***, which is an indication that profile-guided optimization of the benchmark must take these variations into account.

In general, increasing the number of functions selected for specialization also decreases the performance gap relative to whole-program specialization. ***povray*** is the only exception where the performance decreases from E05-S05 to E02-S02. Code inspection revealed that the code produced for whole-program specialization has fewer `nop` instructions than the code produced with selective specialization. `nop` instructions are added to guarantee alignment of the target of branches or to avoid pipeline hazards. It is not clear why the `nop` instructions were added/left by the compiler. Nonetheless, the difference in code alignment produced significant differences in the instruction-cache performance: E02-S02 execution reports 15% more *L1-icache-load-misses* when specializing functions that added `nop`. Removing the specialization attribute from two of these functions — *Intersect_Light_Tree* and *DNoise* — resulted in E02-S02 speedup of 2.10%, compared to the E05-S05 speedup of 1.95%.

D. DASS Specialization Speedup

The results in Section IV-C indicate that there is significant optimization that can be achieved by specializing individual functions. To answer ③, this section contrasts the specialization of selected functions statically (Section IV-C2) to the dynamic specialization of the same functions in the DASS prototype implementation. Only functions in E05-S05 are used for specialization because this criteria presents a middle ground between specialization speedup and the number of targeted functions. Two benchmarks — ***namd*** and ***nab*** — are excluded from this comparison because limitations in the DASS prototype prevent it from specializing functions for those programs.

The results in Figure 4 show that **D-Block** and **D-Start** perform equally for all evaluated benchmarks. Blocking can improve over startup by avoiding extraneous compilations of functions that are not executed for certain benchmark workloads. However, for the analyzed benchmarks all selected functions are called on each workload, minimizing the difference between the two approaches. Asynchronous compilation has a noticeable slowdown when performed on ***x264*** and ***imagick***. Two factors contribute to slowdown. First, while the functions are being compiled the non-sub-target-specialized AOT-compiled code is used and the difference between target and sub-target-specialized code can be significant (Section IV-C). This difference is

TABLE III: Percent of total benchmark time for initializing the DASS runtime and JIT compiling the E05-S05 criteria selected functions.

Benchmark	DASS Initialization	JIT Compilation
perlbench	0.0011%	0.088%
deepsjeng	0.0018%	0.054%
leela	0.000051%	0.0067%
povray	0.000055%	0.0074%
imagick	0.0058%	0.46%
lbm	0.002%	0.035%
x264	0.0026%	0.18%

the cause for ***imagick***, where one selected function executes a single call over the entire workload, and as such never executes specialized code when compiled asynchronously. The second factor is that, if two threads — an application thread and a JIT runtime thread — access the JIT table, synchronization is required to ensure coherent access. Instructions required for synchronization, such as memory fences, incur overhead.

For the benchmarks used in this evaluation, the advantage of asynchronous compilation is limited because their compilation time is not significant. The percentage of total program time to set up the JIT (register IR & Symbols) and perform compilation during execution is shown in Table III. When the compilation time of the function is low, it is beneficial to wait for the compilation to complete.

Although the results in Figure 4 indicate an affirmative answer to ③, if we consider the best-performing variant **D-Block**, the figure shows mixed results in contrast to static specialization speedups. There are instances where DASS is close or equal to static speedup (***lbm***, ***x264***), instances with substantial degradation in speedup (***imagick***) and multiple instances where DASS performs worse than if no specialization was applied (***perlbench***, ***deepsjeng***, ***leela*** and ***povray***). These results could be attributed to differences in code generation or missed optimizations, which arise from compiling the functions in isolation from their original context in the application code. However, an inspection of the code generated by DASS and by the AOT compiler pointed to other differences. Thus, in the spirit of always measuring one level deeper [35], the following section presents an in-depth study of the sources of overhead in DASS.

E. A Study of DASS Overheads

As discussed in Section III-C, there are four main sources of overhead in the current DASS implementation: 1) Compilation overhead (Section III-C1): the cost of performing sub-target specialization at run-time; 2) JIT runtime initialization overhead (Section III-C2): the cost of setting up the ORC and LLVM's components for JITing; 3) Indirection Overhead (Section III-C3): introduced by the trampoline structure used as a dynamic-dispatch mechanism that enables the AOT-compiled code to call JITed functions; and 4) Lower Code & Data Locality Overhead (Section III-C4): caused by the placement of JITed functions potentially far and separate in memory from the function's original context (surrounding functions and variables). Although the access to symbols defined in the AOT-compiled code also undergoes indirection, its effects are mostly observable as poor locality. For the evaluated applications, as Table III shows, both the compilation and initialization overheads are low, thus they are not the main sources of overhead that contribute to the significant slowdown observed in some applications (*e.g.* ***deepsjeng*** and ***povray***).

1) Indirection Overhead: To understand the impact of the JIT trampoline indirection, two modified versions of DASS are contrasted with the static specialized code (**Static**) and **D-Block**: 1) **Indirect**: there is no run-time compilation; use the JIT trampoline to call

TABLE IV: Benchmark metrics comparing **Static** and **D-Block** specialization for E05-S05. Normalized metrics where **D-Block** differs from **Static** by more than 10% are identified with Green if less than, or Red if greater than. DERAT is a small buffer that caches effective to real address translations from the dTLB.

Metric	***perlbench***		***deepsjeng***		***leela***		***povray***		***imagick***		***lbm***		***x264***	
D-Block / Static	1.041x		1.197x		1.076x		1.383x		1.125x		1.000x		1.050x	
	Static	**D-Block**	**Static**	**D-Block**	**Static**	**D-Block**	**Static**	**D-Block**	**Static**	**D-Block**	**Static**	**D-Block**	**Static**	**D-Block**
Instructions	3814B	1.06x	2170B	1.10x	2873B	1.14x	4714B	1.32x	3201B	1.15x	1055B	1.00x	2756B	1.08x
Cycles	1626B	1.04x	1178B	1.12x	1879B	1.08x	2338B	1.37x	1085B	1.11x	638B	0.99x	1575B	1.04x
Instruction Count Normalized Metrics (X / Instructions)														
Branch Miss	$5.4e^{-4}$	$5.2e^{-4}$	$2.4e^{-3}$	$2.2e^{-3}$	$5.3e^{-3}$	$4.6e^{-3}$	$8.6e^{-4}$	$7.6e^{-4}$	$3.9e^{-4}$	$3.5e^{-4}$	$2.5e^{-5}$	$2.5e^{-5}$	$4.2e^{-4}$	$3.8e^{-4}$
Branch Load Miss	$2.8e^{-4}$	$2.8e^{-4}$	$7.4e^{-4}$	$7.4e^{-4}$	$1.6e^{-3}$	$1.4e^{-3}$	$3.8e^{-4}$	$5.1e^{-4}$	$1.2e^{-4}$	$1.1e^{-4}$	$8.3e^{-6}$	$8.5e^{-6}$	$1.6e^{-4}$	$1.5e^{-4}$
iCache Miss	$1.6e^{-3}$	$1.7e^{-3}$	$7.3e^{-5}$	$4.7e^{-4}$	$4.4e^{-5}$	$3.9e^{-5}$	$7.0e^{-4}$	$1.9e^{-3}$	$3.7e^{-7}$	$1.2e^{-5}$	$2.1e^{-6}$	$4.6e^{-6}$	$5.6e^{-4}$	$6.0e^{-4}$
dTLB Miss	$1.7e^{-8}$	$1.7e^{-8}$	$1.0e^{-4}$	$9.2e^{-5}$	$4.1e^{-7}$	$3.8e^{-7}$	$2.5e^{-10}$	$2.5e^{-10}$	$6.8e^{-9}$	$7.0e^{-9}$	$7.2e^{-6}$	$7.2e^{-6}$	$4.5e^{-8}$	$3.1e^{-8}$
iTLB Miss	$6.3e^{-6}$	$4.8e^{-7}$	$4.4e^{-7}$	$9.9e^{-5}$	$3.1e^{-7}$	$1.3e^{-7}$	$1.2e^{-6}$	$2.7e^{-6}$	$8.2e^{-10}$	$2.7e^{-8}$	$4.9e^{-9}$	$1.5e^{-8}$	$5.3e^{-8}$	$4.8e^{-8}$
Cycle Count Normalized Metrics (X / Cycles)														
L1 Miss Stalls	0.031	0.030	0.017	0.018	0.012	0.012	0.039	0.033	$1.6e^{-3}$	$2.0e^{-3}$	$5.1e^{-5}$	$6.9e^{-5}$	$8.4e^{-3}$	$8.3e^{-3}$
FE Stall	0.019	0.020	0.022	0.023	0.042	0.040	0.013	0.018	$5.7e^{-3}$	$7.3e^{-3}$	0.028	0.027	$5.4e^{-3}$	$6.3e^{-3}$
BE Stall	0.535	0.553	0.594	0.571	0.538	0.541	0.732	0.670	0.717	0.718	0.746	0.745	0.698	0.685
DERAT Miss Stall	$1.3e^{-4}$	$1.3e^{-4}$	$2.5e^{-6}$	$2.8e^{-6}$	$4.4e^{-6}$	$7.0e^{-6}$	$4.6e^{-8}$	$1.6e^{-6}$	$1.1e^{-6}$	$2.4e^{-6}$	$1.7e^{-5}$	$1.5e^{-5}$	$2.3e^{-6}$	$5.0e^{-6}$
DERAT Miss	$9.8e^{-4}$	$1.0e^{-3}$	$3.6e^{-4}$	$3.4e^{-4}$	$7.4e^{-5}$	$8.9e^{-5}$	$1.1e^{-6}$	$6.6e^{-5}$	$1.1e^{-5}$	$1.9e^{-5}$	$4.1e^{-5}$	$4.2e^{-5}$	$6.2e^{-5}$	$1.0e^{-4}$
dTLB Miss Stall	$1.4e^{-6}$	$1.4e^{-6}$	$1.2e^{-5}$	$7.3e^{-5}$	$1.6e^{-5}$	$1.3e^{-5}$	$1.1e^{-7}$	$9.6e^{-8}$	$1.0e^{-6}$	$9.3e^{-7}$	$1.1e^{-4}$	$1.0e^{-4}$	$5.1e^{-6}$	$5.4e^{-6}$

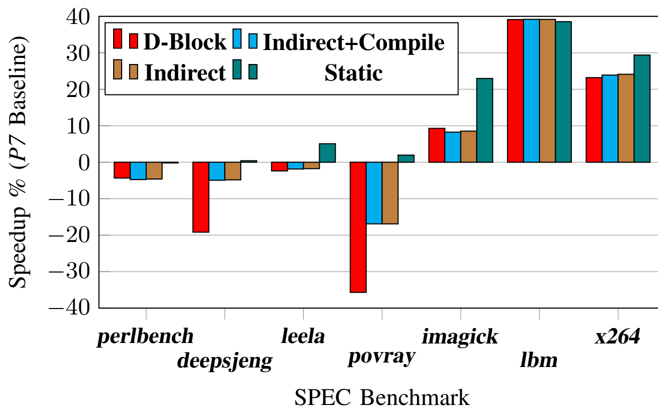

Fig. 5: Contrasting speedup for E05-S05 of static specialization (**Static**), dynamic specialization (**D-Block**), and two modifications of DASS: **Indirect**, that uses the JIT trampoline to call AOT-compiled code; and **Indirect+Compile**, that also calls AOT-compiled code but still compiles the function at runtime.

the AOT-compiled function code instead of the JITed code; and 2) **Indirect+Compile**: perform compilation on the first invocation of the selected functions, but then use the trampoline to call the AOT-compiled function code. In both versions, the AOT-compiled code is sub-target-specialized via the same function attribute used in **Static**. Results in Figure 5 indicate that trampoline indirection varies on whether it is a significant overhead. While Table IV shows the trampoline overhead mostly comes from extra instructions. Both **Indirect** and **Indirect+Compile** are significantly faster than DASS for ***deepsjeng*** and ***povray***, instances where the AOT-specialized code is faster than the JITed code. In comparison, the equal slowdown for **Indirect+Compile** and **D-Block** with ***imagick*** and ***x264***, indicates that for those benchmarks the trampoline mechanism is the main overhead. This is supported by function profiles, wherein both benchmarks frequently call their target functions, at an average rate of a call per roughly 11 and 25 cycles of the selected functions.

2) Lower Code & Data Locality Overhead: The sources of poor locality in DASS prototype are multi-faceted. When a JITed function is loaded at run-time, it might be placed far in memory — *e.g.* on different pages — relative to the AOT-compiled code that calls it. Code located on different pages requires multiple entries in the iTLB and may exhibit poor instruction-cache locality. The results in Table IV indicate that this is the case for ***povray***, ***deepsjeng***, and ***imagick*** because these applications experience significantly more instruction cache and iTLB misses with **D-Block** than with **Static**. Moreover, placing the JIT function in a different memory location may have an effect on the prediction of existing branches — *e.g.* it may introduce branch aliasing. Such effects could explain the 34% increase in branch load misses in ***povray***.

JITed code can also be impacted by lower locality when accessing data because the JIT runtime cannot make assumptions on where in memory the JITed code will be loaded, it must be conservative and not use memory instructions that encode small relative offsets — *e.g.* 32 bits in the medium code model or 16 bits in the small code model. Instead, the JITed code accesses symbols in the AOT-compiled program through a table — generated by the JIT runtime (Section III-B3). Thus, every access to AOT-compiled code symbols in the JITed code goes through an indirection. The indirection adds overhead in terms of more instructions per access. More importantly, because the table is placed on separate memory pages from both the JITed and AOT-compiled code, the indirection may add more overhead in terms of address translation. A comparison of **D-Block** with **Static** in Table IV indicates that: 1) there are more stalls due to DERAT[4] misses ranging from 12% in ***deepsjeng*** up to 34.8× in ***povray***; 2) DERAT misses are 81% higher in ***imagick*** and 60× higher in ***povray***; 3) ***deepsjeng*** exhibits 6× more dTLB misses.

3) Addressing the Overhead in the DASS Prototype: The results from Section IV-E2 answer ④, indicating that the observed overhead is caused by a mix of indirection and lower code & data locality introduced by the placement of JITed code relative to AOT-compiled code. For indirection, simplifying the calls to JITed code, by rewriting call sites instead of executing the trampoline structure, can reduce overhead, however, this is not feasible for many developers. While instruction locality is hard to improve, changes can improve the

[4]A small buffer that caches effective-to-real address translations from the dTLB.

locality of data. In particular, the indirection to access data can be avoided by passing the addresses of symbols as arguments to the JITed function. However, this solution is dependent on the Application Binary Interface and would only work for functions that access a small number of symbols, otherwise the arguments would be passed through the stack and cause a similar locality issue. Moreover, passing addresses to functions may affect the results of escape analysis and reference analysis with detrimental effects on performance. Another possibility is to employ an inspector/executor approach that would observe the address range where the JITed code was loaded and, if the distance to accessed symbols permits, replace indirect loads with (direct) offset-based loads. The efficacy and trade-offs of such solutions are left for future investigation.

V. Related Works

Whole program AOT specialization is often avoided in exchange for more generically compiled binaries which can execute on the various sub-targets of a given CPU architecture. However, work has been done to reduce the build-time cost of AOT specialization by decreasing its scope. Multi-versioning, like that implemented in GCC [36], allows for applying explicit specialization to target functions. Managed by a dynamic check this method avoids issues of JIT compilation and the locality of JITed code. However, unlike DASS, it requires saving each version of the function code to the binary and is limited to the sub-targets explicitly defined in the source code.

Dynamic code compilation for static languages has been a persistent point of interest in prior research. Dynamic compilation has two major approaches: holistic and selective. A holistic approach targets the entire program for recompilation during execution, while a selective approach narrows the scope to "high value" segments of the code.

The choice of which segments to target for specialization influences the possible optimizations that dynamic compilation can perform and the corresponding cost. DyC [37] focuses on individual variables, recompiling the code blocks where these variables appear to generate binaries that can treat the variables as run-time constants. ADAPT [38] identifies loop nests that lack function calls or I/O, creating multiple experimental versions of the code and picking the best performing version. Azure [39] targets Single-Entry-Single-Exit regions of binary code that demonstrate sufficient parallelism, producing modified binaries that take advantage of new hardware constructs. Castanos et al.'s work [29] and ExanaDBT [40] focus on recompiling entire functions, allowing for easier insertion of recompiled code into the host program through call modification or through a trampoline to a dynamic library. DASS employs function-level specialization and thus is not limited to optimizations that target specific variable usage or loops as DyC and ADAPT. Instead of raising binary code to an IR-form like in Azure and ExanaDBT, DASS saves the IR of selected segments into the binary, thus it preserves more static information. Castanos et al.'s work [29] is the most similar work to DASS. However, the system proposed by Castanos et al. is designed to use JIT technology while the DASS implementation used a JIT compiler as a convenient way to produce a prototype, as DASS can be realized without a JIT.

Identification of recompilation targets requires balance between the amount of information required for the decision, and the overhead of retrieving that information and making the decision. DyC [37] and Tick C [41] makes use of a manually applied annotation attached to each code line that identifies it for dynamic compilation. Forgoing manual identification, Calpa [25] extends DyC by applying the system's annotations automatically, requiring a profile of the program to built in a previous run of the program to inform its decisions. Unlike DyC and Tick C, DASS can specialize segments at the level of functions, thus the selection can be done manually via function attributes or automatically by the compiler. In addition, the runtime component of DASS can be extended to only apply sub-target specialization to candidates that pass a set of run-time cost/benefit criteria.

Other methods also focus on improving performance through a combination of architectural and program behavior information, but they forgo a selective approach to the recompilation targets. BOLT [42] optimizes code for data-center applications by collecting offline profiles to inform dynamic recompilation that is performed over the entire program to improve cache efficiency through binary layout reordering. PROPELLER [43] distributes a compilation process similar to BOLT's, but it also reduces the, potentially significant, overhead incurred by profiling. Lightning BOLT [44] develops a similar process to reduce compilation costs by enabling parallelism within the most costly optimization. It transforms the approach into a selective one by performing analysis on the respective reduced cost and retained value of only compiling certain functions. Different than BOLT, DASS does not require profiling data as candidates for dynamic specialization can be identified via other mechanisms, *e.g.* static analysis or run-time cost/benefit heuristics. Moreover, the goals of BOLT are orthogonal to DASS's, thus they can be employed together.

VI. Conclusion

This work evaluated the feasibility of a compiler-based system to apply dynamic adaptive sub-target specialization (DASS). Results from an in-depth evaluation indicate that there is significant performance gains that can be achieved through static sub-target specialization. The results further indicate that it possible to attain similar results by applying dynamic specialization at run-time. Furthermore, empirical evidence indicates that it is sufficient to specialize only a fraction of the application code in order to achieve the majority of the whole-application specialization gains. A detailed analysis of the overheads observed in a prototype implementation of DASS in LLVM reveals that, although effective, a JIT-enabled approach requires careful consideration of what functions should be selected for sub-target specialization. Moreover, the detailed overhead analysis identified sources of overhead that are inherent to a JIT-enabled approach.

Further improvements to DASS can be made. Automatic selection of functions can improve the usability of the system. This selection can be informed both by static analysis at compile time, wherein the fat-binary size is decided, and dynamic analysis during execution, wherein compilation of less frequently executed functions, for a given workload, can be avoided. Other areas of improvement include refinement of the JIT compilers optimization pipeline. Currently the default compiler pipeline is used, however this includes passes with no impact on sub-target specialization, which can be performed safely on fat-binary IR prior to serialization. Adding optimization passes that take advantage of the execution time information available to DASS is another future consideration. These can be used to further optimize segments by taking advantage of the executing hardware's capabilities (*e.g.* shared-memory parallelism). Finally, the aforementioned techniques to reduce data locality overhead for JITed code can improve DASS performance relative to static specialization.

References

[1] M. Cornea, "Intel AVX-512 instructions and their use in the implementation of math functions," 2015.

[2] A. Strey and M. Bange, "Performance analysis of intel's MMX and SSE: A case study," in Euro-Par 2001 Parallel Processing: 7th International Euro-Par Conference Manchester, UK, August 28–31, 2001 Proceedings 7. Springer, 2001, pp. 142–147.

[3] K. Diefendorff, P. K. Dubey, R. Hochsprung, and H. Scale, "Altivec extension to powerpc accelerates media processing," IEEE Micro, vol. 20, no. 2, pp. 85–95, 2000.

[4] L. Eisen, J. Ward, H.-W. Tast, N. Mading, J. Leenstra, S. M. Mueller, C. Jacobi, J. Preiss, E. M. Schwarz, and S. R. Carlough, "Ibm power6 accelerators: Vmx and dfu," IBM Journal of Research and Development, vol. 51, no. 6, pp. 1–21, 2007.

[5] C. Lomont, "Introduction to intel advanced vector extensions," Intel, Tech. Rep., 2011.

[6] J. E. Moreira, K. Barton, S. Battle, P. Bergner, R. Bertran, P. Bhat, P. Caldeira, D. Edelsohn, G. Fossum, B. Frey et al., "A matrix math facility for Power ISA(TM) processors," arXiv preprint arXiv:2104.03142, 2021.

[7] Arm® Architecture Reference Manual Armv8, for Armv8-A Architecture Profile, Arm Limited, Jan. 2021.

[8] Intel® Architecture Instruction Set Extensions and Future Features Programming Reference, Intel Corporation, Feb. 2021.

[9] S. Kumar, Y. Wang, C. Young, J. Bradbury, N. Kumar, D. Chen, and A. Swing, "Exploring the limits of concurrency in ml training on google tpus," Proceedings of Machine Learning and Systems, vol. 3, pp. 81–92, 2021.

[10] I. Magaki, M. Khazraee, L. V. Gutierrez, and M. B. Taylor, "Asic Clouds: Specializing the Datacenter," in 2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA). Seoul, South Korea: IEEE, 2016, pp. 178–190.

[11] J. Nickolls and W. J. Dally, "The gpu computing era," IEEE micro, vol. 30, no. 2, pp. 56–69, 2010.

[12] G. Krylov, P. Jelenković, M. Thom, G. W. Dueck, K. B. Kent, Y. Manton, and D. Maier, Ahead-of-Time Compilation in Eclipse OMR on Example of WebAssembly. USA: IBM Corp., 2021, p. 237–243.

[13] C.-S. Wang, G. Perez, Y.-C. Chung, W.-C. Hsu, W.-K. Shih, and H.-R. Hsu, "A method-based ahead-of-time compiler for android applications," in Proceedings of the 14th international conference on Compilers, architectures and synthesis for embedded systems, 2011, pp. 15–24.

[14] Faraboschi, Paolo and Fisher, Joseph A and Young, Cliff, "Instruction scheduling for instruction level parallel processors," Proceedings of the IEEE, vol. 89, no. 11, pp. 1638–1659, 2001.

[15] J. Bucek, K.-D. Lange, and J. v. Kistowski, "Spec cpu2017: Next-generation compute benchmark," in Companion of the 2018 ACM/SPEC International Conference on Performance Engineering, 2018, pp. 41–42.

[16] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in International Symposium on Code Generation and Optimization, 2004. CGO 2004. San Jose, CA, USA: IEEE, 2004, pp. 75–86.

[17] Kretschmer, Tobias and Claussen, Jörg, "Generational Transitions in Platform Markets — The Role of Backward Compatibility," Strategy Science, vol. 1, no. 2, pp. 90–104, 2016.

[18] K. Sangani, "Computer says no [software compatibility]," Engineering & Technology, vol. 6, no. 9, pp. 76–77, 2011.

[19] Sammet, Jean E, "The Early History of COBOL," History of Programming Languages, vol. 1, pp. 199–243, 1978.

[20] Backus, John, "The history of Fortran I, II, and III," ACM Sigplan Notices, vol. 13, no. 8, pp. 165–180, 1978.

[21] Ritchie, Dennis M., "The Development of The C Language," ACM Sigplan Notices, vol. 28, no. 3, pp. 201–208, 1993.

[22] B. Stroustrup, The C++ Programming Language, 4th ed. Boston, Massachusetts, United States: Addison-Wesley Professional, 2013.

[23] M. Thom, G. W. Dueck, K. Kent, and D. Maier, "A survey of ahead-of-time technologies in dynamic language environments," in Proceedings of the 28th Annual International Conference on Computer Science and Software Engineering, 2018, pp. 275–281.

[24] N. Reijers and C.-S. Shih, "Ahead-of-Time Compilation of Stack-Based JVM Bytecode on Resource-Constrained Devices," in Transactions on Sensor Networks. New York, NY, United States: Association for Computing Machinery, 2017, pp. 84–95.

[25] M. Mock, C. Chambers, and S. Eggers, "Calpa: a tool for automating selective dynamic compilation," in Proceedings 33rd Annual IEEE/ACM International Symposium on Microarchitecture. MICRO-33 2000. Monterey, CA, USA: IEEE, 2000, pp. 291–302.

[26] Aycock, John, "A Brief History of Just-In-Time," ACM Computing Surveys (CSUR), vol. 35, no. 2, pp. 97–113, 2003.

[27] A. Krall, "Efficient JavaVM Just-In-Time Compilation," in Proceedings. 1998 International Conference on Parallel Architectures and Compilation Techniques (Cat. No. 98EX192). Washington, DC, United States: IEEE, 1998, pp. 205–212.

[28] A. Gal, B. Eich, M. Shaver, D. Anderson, D. Mandelin, M. R. Haghighat, B. Kaplan, G. Hoare, B. Zbarsky, J. Orendorff et al., "Trace-based Just-In-Time Type Specialization for Dynamic Languages," ACM Sigplan Notices, vol. 44, no. 6, pp. 465–478, 2009.

[29] D. Nuzman, R. Eres, S. Dyshel, M. Zalmanovici, and J. Castanos, "Jit technology with c/c++ feedback-directed dynamic recompilation for statically compiled languages," ACM Transactions on Architecture and Code Optimization (TACO), vol. 10, no. 4, pp. 1–25, 2013.

[30] LLVM Developer Group, "Orcv2: Design and implementation," https://llvm.org/docs/ORCv2.html, 2019, accessed: 2023-04-25.

[31] R. Ierusalimschy, L. H. De Figueiredo, and W. C. Filho, "Lua — An Extensible Extension Language," Software: Practice and Experience, vol. 26, no. 6, pp. 635–652, 1996.

[32] V. Vasilev, P. Canal, A. Naumann, and P. Russo, "Cling–the new interactive interpreter for root 6," in Journal of Physics: Conference Series, vol. 396, IOP Publishing. New York, NY, USA: IOP Publishing, 2012, p. 052071.

[33] M. Wilde, M. Hategan, J. M. Wozniak, B. Clifford, D. S. Katz, and I. Foster, "Swift: A language for distributed parallel scripting," Parallel Computing, vol. 37, no. 9, pp. 633–652, 2011.

[34] SPEC® Open Systems Group, "SPEC CPU®2017 run and reporting rules," https://www.spec.org/cpu2017/Docs/runrules.html, 2020, accessed: 2023-08-17.

[35] J. Ousterhout, "Always measure one level deeper," Commun. ACM, vol. 61, no. 7, p. 74–83, jun 2018. [Online]. Available: https://doi.org/10.1145/3213770

[36] S. Tallam, "Function multiversioning," https://gcc.gnu.org/wiki/FunctionMultiVersioning, 2013, accessed: 2023-08-17.

[37] B. Grant, M. Mock, M. Philipose, C. Chambers, and S. J. Eggers, "DyC: an expressive annotation-directed dynamic compiler for C," Theoretical Computer Science, vol. 248, no. 1-2, pp. 147–199, 2000.

[38] M. J. Voss and R. Eigemann, "High-level adaptive program optimization with ADAPT," in Proceedings of the eighth ACM SIGPLAN symposium on Principles and practices of parallel programming. New York, NY, USA: Association for Computing Machinery, 2001, pp. 93–102.

[39] E. Yardimci and M. Franz, "Mostly static program partitioning of binary executables," ACM Transactions on Programming Languages and Systems (TOPLAS), vol. 31, no. 5, pp. 1–46, 2009.

[40] Y. Sato, T. Yuki, and T. Endo, "ExanaDBT: A Dynamic Compilation System for Transparent Polyhedral Optimizations at Runtime," in Proceedings of the Computing Frontiers Conference. New York, NY, United States: Association for Computing Machinery, 2017, pp. 191–200.

[41] D. R. Engler, W. C. Hsieh, and M. F. Kaashoek, "C: A language for high-level, efficient, and machine-independent dynamic code generation," in Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages, 1996, pp. 131–144.

[42] M. Panchenko, R. Auler, B. Nell, and G. Ottoni, "Bolt: a practical binary optimizer for data centers and beyond," in 2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO), IEEE. Washington, DC, USA: IEEE Press, 2019, pp. 2–14.

[43] H. Shen, K. Pszeniczny, R. Lavaee, S. Kumar, S. Tallam, and X. D. Li, "Propeller: A profile guided, relinking optimizer for warehouse-scale applications," in Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2, 2023, pp. 617–631.

[44] M. Panchenko, R. Auler, L. Sakka, and G. Ottoni, "Lightning BOLT: powerful, fast, and scalable binary optimization," in Proceedings of the 30th ACM SIGPLAN International Conference on Compiler Construction, 2021, pp. 119–130.