DASS: Dynamic Adaptive Sub-Target Specialization

by

Tyler Gobran

A thesis submitted in partial fulfillment of the requirements for the degree of

Master of Science

Department of Computing Science

University of Alberta

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 thesis provides an overview of existing sub-target specialization methods and demonstrates that challenges to existing methods 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.

Preface

DASS, the central idea of this thesis, has been submitted for publication as T. Gobran, J.P.L. de Carvalho, Q. Pham, J.N. Amaral, C. Barton, N. Ivanovic, "DASS: Dynamic Adaptive Sub-Target Specialization". I was responsible for significant aspects of the concept of the DASS framework, and the main design, construction, and evaluation of an LLVM prototype implementation. J.P.L. de Carvalho provided guidance for the design and evaluation and technical help for the implementation. Q. Pham helped with the technical implementation and with performing a preliminary evaluation that was used to guide the formal evaluation presented in this thesis. J.N. Amaral was the supervisory author and contributed to guiding the experimental evaluation and editing the resulting manuscript. C. Barton and N. Ivanovic were IBM collaborators, that participated in technical discussions on the implementation and evaluation results.

To Enlik

You are the inspiration for everything that I do.

Acknowledgements

I would like to thank my co-supervisor's, Dr. J. Nelson Amaral and João Paulo Labegalini de Carvalho for their support and guidance throughout the process of completing this thesis. I would also like to thank Quinn Pham for his help in implementing the prototype for DASS.

This research is funded by the IBM Center for Advanced Studies (CAS) Canada and by the Natural Science and Engineering Research Council (NSERC) of Canada through its Collaborative Research and Development (CRD) program. It has been completed in collaboration with Kit Barton and Nemanja Ivanovic from IBM Canada.

Contents

1 Introduction **1**

2 Background **4**
2.1 Independent Software Vendor (ISV) 4
2.2 Ahead-Of-Time (AOT) Compilation 5
2.3 Just-In-Time (JIT) Compilation 5

3 Sub-Target Specialization **7**
3.1 Static Specialization 7
3.1.1 Sub-Target Builds 9
3.1.2 Static Binary Recompilation 9
3.1.3 Shared Libraries 10
3.2 Dynamic Specialization 11
3.2.1 Interpretation 12
3.2.2 Function Specialization 12

4 Dynamic Adaptive Sub-Target Specialization (DASS) **16**
4.1 Core Design 16
4.1.1 Fat Binary Creation 17
4.1.2 Dynamic Code Redirection 17
4.1.3 Symbol Resolution 18
4.1.4 Dynamic Sub-Target Specialization 18
4.2 DASS Prototype Implementation 18
4.2.1 Extensions to Clang 19
4.2.2 IR Cloning and Dynamic-Dispatch Insertion Pass . . . 19
4.2.3 JIT Runtime 21
4.2.4 Extensions to LLD 22
4.3 DASS Overhead 23
4.3.1 Compilation Overhead 23
4.3.2 Initialization Overhead 24
4.3.3 Indirection Overhead 24
4.3.4 Optimization Limitation Overhead 25
4.3.5 Lower Code & Data Locality Overhead 25

5 Evaluation **26**
5.1 Experimental Setup and Methodology 26
5.1.1 DASS Prototypes 27
5.1.2 Specialization & Selection Methods 29
5.2 Static Specialization 31
5.2.1 Whole-Program Specialization 31
5.2.2 Selective-Function Specialization 36
5.3 DASS Specialization 41

5.4 DASS Overhead 45
5.4.1 Initialization & Compilation Overhead 46
5.4.2 Indirection Overhead 49
5.4.3 Optimization Limitation Overhead 53
5.5 DASS Metric Analysis 55
5.5.1 Lower Code & Data Locality Overhead 59
5.6 Summary 60
5.6.1 Function Selection Takeaways 62

6 Related Work **65**
6.1 Dynamic Compilation Methods 65
6.1.1 Holistic Dynamic Compilation 66
6.1.2 Selective Dynamic Compilation 67
6.2 DASS Expansion Works 68

7 Conclusion **70**

References **72**

Appendix A Function Selection **78**

List of Tables

4.1 DASS Prototype Overhead Sources 23

5.1 Evaluation Machine Configurations 27
5.2 Machine-9 Function Selections 30
5.3 Machine-10 Function Selections 30
5.4 Machine-9 Initialization & Compilation Overhead 47
5.5 Machine-10 Initialization & Compilation Overhead 48
5.6 Machine-9 Selected Function Indirect Call Occurence 51
5.7 Machine-10 Selected Function Indirect Call Occurence 51
5.8 Machine-9 DASS Performance Metrics 57
5.9 Machine-10 DASS Performance Metrics 58

A.1 Machine-9 E10-S10 Functions 79
A.2 Machine-9 E05-S05 Functions 80
A.3 Machine-9 E02-S02 Functions 81
A.4 Machine-10 E10-S10 Functions 83
A.5 Machine-10 E05-S05 Functions 84
A.6 Machine-10 E02-S02 Functions 85

List of Figures

3.1 Static Specialization Methods 8
3.2 Dynamic Specialization Methods 13

4.1 JIT Call Trampoline 20

5.1 DASS Prototype Diagram 28
5.2 Machine-9 Whole-Program Specialization Speedup 32
5.3 Machine-10 Whole-Program Specialization Speedup 34
5.4 Machine-9 Static Selective Function Speedup 36
5.5 Machine-10 Static Selective Function Speedup 39
5.6 Machine-9 DASS Speedup 42
5.7 Machine-10 DASS Speedup 43
5.8 Machine-9 Indirection Overhead 50
5.9 Machine-10 Indirection Overhead 52
5.10 Machine-9 Optimization Limitation Overhead 54
5.11 Machine-10 Optimization Limitation Overhead 55

Chapter 1

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 [12]–[14], [32], [37], [52]. 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) [5], [23]. Acceleration through specialization is not new but it is a growing trend [28], [34], [39]. 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) [27], [59]. An alternative to AOT compilation is **Just-In-Time** (JIT) compilation that delays optimization and code generation until application runtime (Section 2.3). 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 [18]. 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 thesis 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 [10] indicate that dynamic specialization can enable ISVs to deliver close to the fully-specialized performance. This thesis makes the following contributions:

- An overview of the applicability and downsides of existing static and dynamic sub-target specialization techniques.
- An in-depth performance evaluation of the SPEC CPU 2017 benchmarks, which shows that specializing only a small fraction of the application 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 implementation of DASS using LLVM [29] that enables close to AOT-specialized performance for some benchmarks from the SPEC CPU 2017 suite.

The remainder of this thesis is organized as follows. Chapter 2 presents background concepts and terminology adopted throughout the thesis. Chapter 3 explains the concept of sub-target specialization and provides an overview of existing methods for performing specialization on program code. Chapter 4 presents the main ideas behind DASS and a description of the proof-of-concept implementation in LLVM. Chapter 5 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. Chapter 6 contrasts related works with DASS. The conclusions of this study are discussed in Chapter 7.

Chapter 2

Background

This thesis focuses on optimization benefits enabled by **Sub-Target Specialization** (Chapter 3), through which a program is optimized 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 2.2) 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 2.3) 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.

2.1 Independent Software Vendor (ISV)

An Independent Software Vendor (ISV) develops and distributes software as pre-packaged binaries to clients outside of their own organization. Distributing large software projects, that must meet client expectations of functionality, leads to long and costly production pipelines that encourage cost saving wherever possible. Considering this investment, maintaining additional software versions requires clear business value. Builds for different CPU targets expand the reach

of the software to new hardware platforms and clients. While, sub-target builds, designed to achieve better performance, do not provide the same growth in reach and their optimizations can vary in value (Section 5.2). Thus, ISVs tend to avoid sub-target specialization, limiting hardware optimization on software that can be functionally ubiquitous on client computer systems.

2.2 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.) [7], [46], [47], [53] 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 [35], [45], [55]. 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 and quickly.

2.3 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 [6]. 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 [25]. 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 [19]. Ideally, the compilation overhead is offset by the speedup on future executions, which payout if the JIT-compiled code re-executes often.

Chapter 3

Sub-Target Specialization

Most optimizing compilers, by default, generate code that is generic to a given CPU architecture, or **target**. 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 [37]. 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 [26], [48]. 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.

3.1 Static Specialization

In the context of this thesis, **static specialization** refers to any sub-target-centred compiler optimization that occurs prior to the execution of the program. Optimization can occur as a part of the original program build process (Sec-

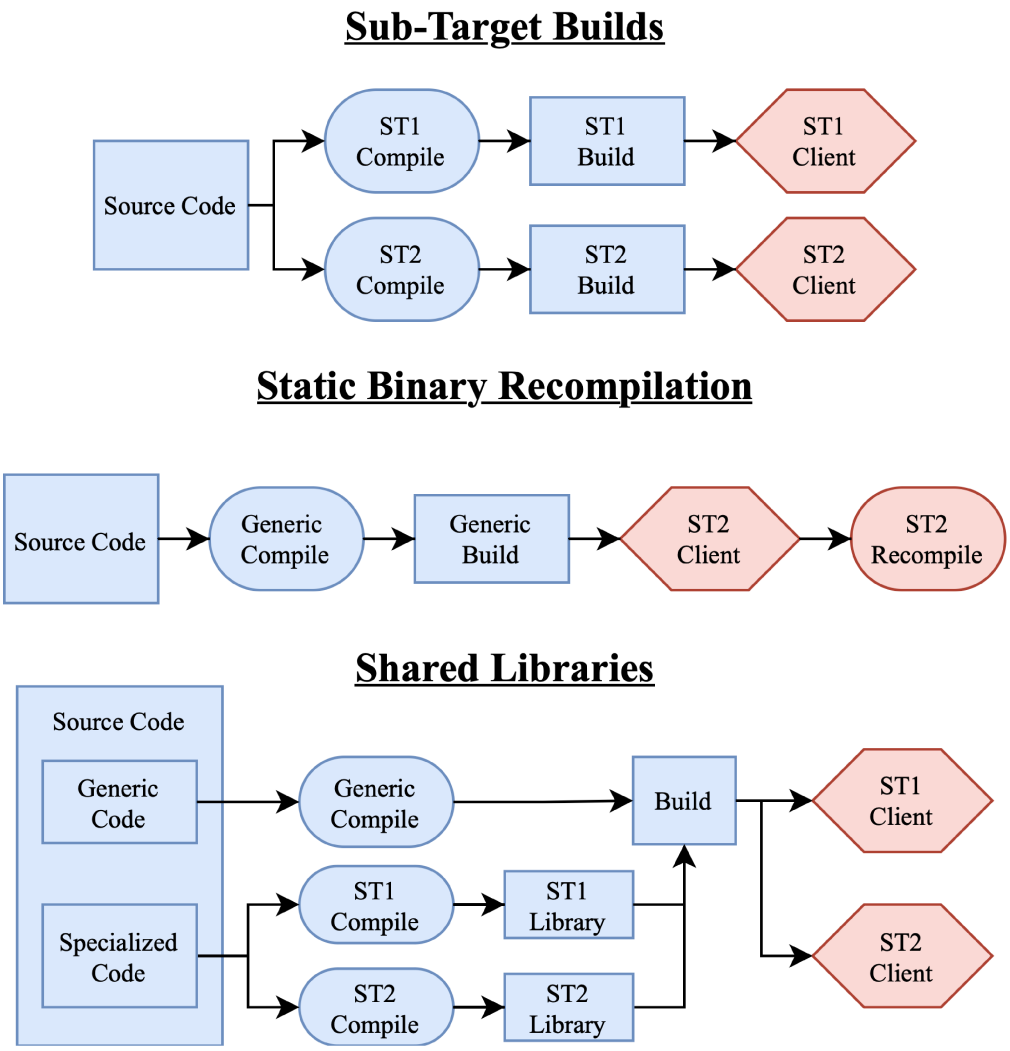

Figure 3.1: Different methods of static sub-target specialization applied on example sub-targets (ST1 and ST2). Stages executed before deployment are shown in Blue and those executed after are in Red.

tion 3.1.1 and Section 3.1.3), or through a later recompilation of the binary on the target hardware (Section 3.1.2). The cost of this optimization occurs once when performed and is independent from program execution, allowing for more aggressive optimization. However, the cost of optimization can still be prohibitive depending on the amount of code that is optimized, as such methods vary in terms of whether the client or vendor pays this optimization cost, and the scope of static specialization for a program. An overview of these methods is shown in Figure 3.1; the following sections explain them in detail.

3.1.1 Sub-Target Builds

The most straightforward method for achieving sub-target specialization is for the software vendor to optimize and maintain separate builds for each sub-target that they support. These full builds can then be distributed to clients based on their respective hardware needs. Separate builds allow for the generated code to target the specific sub-target throughout the compilation process. Furthermore, there is no execution-time compilation overhead and there is no need for additional dynamic mechanisms to identify the sub-target of the executing hardware.

However, these advantages comes at a significant cost for an ISV that wants to maintain builds for each sub-target. Each build requires development resources that impacts the production budget and cost structure of ISVs. For small software this is less impactful, as software build times are manageable. However, for a large software project, the pipeline of compiling and testing the software can become a significant time cost [44], [54], [64], making the effort of supporting additional builds a major resource investment. Furthermore, because sub-target optimization changes the instruction generation for a program [40], bug testing cannot be generalized between different sub-target builds as each can introduce unique compiler bugs. While this problem exists for all specialization methods, those that apply specialization at the scope of the entire program, like sub-target builds, require further testing than methods with a selective and small scope for specialization. Thus, when considering the ISV perspective, sub-target builds are often not a reasonable method to achieve specialization, and as such more complex methods are considered.

3.1.2 Static Binary Recompilation

Given the often prohibitive cost of performing code specialization in the production pipeline of an ISV program, moving that specialization onto the client machine is a promising idea. This can be achieved through binary recompilation [42], [61], the process of decompiling a binary to an optimizable IR format, and recompiling the IR back to a binary with new optimizations. In

this instance, the optimization is focused on the current hardware sub-target on the client machine. This allows the ISV to ship a generic binary while clients who desire more performant code can still attain the benefits of sub-target specialization.

Despite the possible benefits, this method is limited by the capabilities of decompilers. Many decompilers focus on lower-level optimizations and local optimizations that may not include the more extensive code analysis needed for efficient sub-target specialization. One of the current limitations of decompilers is that often they are not able to fully recover type information or access patterns to dynamically allocated memory, thus limiting the information that can be recovered in the regenerated IR and limiting the effectiveness of sub-target specialization [61]. Moreover, some decompilers require a significant time investment to decompile and recompile the program [4]. Further complicating the situation, decompilers can reveal proprietary secrets encoded in the software distributed by an ISV, and as such code may be intentionally obfuscated in the process of generating a binary format [11], hamstringing a decompiler's optimization potential. In some commercial software, contractual clauses may prevent both clients and compiler or hardware vendors from performing decompilation. With these complications, an ISV relying on their clients to recompile code themselves is unlikely. Even if desired, a client could face prohibitive time costs to perform the re-optimization.

3.1.3 Shared Libraries

Reducing the scope of specialization to individual functions or files is another method of reducing the impact of specialization on an ISV production pipeline. Instead of maintaining separate builds for the entire program, functions designated for specialization are extracted to be compiled to a shared library. Building a version of the shared library specialized for each desired sub-target, all of the versions are packed with the program binary. At run-time, the function version specialized for the executing hardware is selected by a simple run-time mechanism. In practice, this method is akin to having a class containing the targeted functions, and creating sub-classes that override them with

sub-target specialized functions.

The reduced scope of specialization in this method reduces the impact on the production pipeline but also reduces the potential optimization benefit and introduces a small selection mechanism overhead. While **Link-Time-Optimization** (LTO) can replace this mechanism with a call to the correct shared library, selecting functions that both benefit from specialization and are significant in terms of program execution is the main concern. Selection is a difficult task, as the benefit from specialization and relative hotness of a function can vary by hardware and workload. Furthermore, the process of extracting code to a shared library and performing the selection mimics actions taken with dynamic function specialization (Section 3.2.2), while losing a significant portion of the dynamic information available to the latter. As such, a shared library optimization strategy is limited to instances where the development team in an ISV has explicit knowledge of what functions will both benefit from optimization across workloads and which will not overwhelm build resources to compile for specialized shared-libraries.

3.2 Dynamic Specialization

Dynamic specialization methods perform sub-target specialization by optimizing for the executing hardware contemporaneous with program execution. Compared to static specialization, dynamic specialization is capable of adapting to new hardware sub-targets automatically, avoiding the need to manually identify the specific sub-target that specialization will target. Furthermore, dynamic methods have the potential to access in-depth metrics on program performance and hardware capabilities, which can be used to further tune optimization. The trade-off, for the availability of dynamic information, is that the time to perform sub-target optimization acts as an overhead on the execution time of the optimized program. Specialization methods must attempt to compensate for this overhead, through a mix of reducing the scale of overhead and performing specialization optimizations that pay-off the cost. The following sub-sections summarize different dynamic specialization methods

(shown in Figure 3.2), focusing on how they reduce overhead by either reducing optimization capabilities (Section 3.2.1) or the scope of program code that is optimized (Section 3.2.2).

3.2.1 Interpretation

Dynamic-typed and interpreted languages, such as Python [49], reduce, and in some instances entirely remove, build-time cost through interpretation. Interpretation reads and executes the original source code or a compiler-produced reduced bytecode format [57], executing through an installed interpreter that is configured to the hardware sub-target. Ideal for quick software testing, interpretation overhead is high [24] compared to AOT compiled code, as such achieving specialization speedup is difficult for this method.

Python is considered a "glue" language that enables the efficient combination of high-performing modules written in other languages. Therefore, for Python, the versatility of run-time interpretation justifies the higher overhead because most of the performance-sensitive computation is coded in higher-performance AOT compiled languages and connected to the Python code through bindings. Attempts to use interpretation for AOT compiled language such as `C/C++` often require reduction to a subset of the full feature set, and delivers performance far below the capabilities of a pre-compiled binary [33]. Thus interpretation of `C/C++`, and other AOT compiled languages, focuses on a lowered version of the language that is closer to machine code to reduce the interpretation overhead [8]. However, this design decision also reduces the optimization capabilities of the interpreter. Thus, there is very limited scope for the use of interpretation to achieve dynamic specialization. Many interpretation systems for AOT languages either ignore performance in the pursuit of other goals (e.g rapid application development) [56] or act as a low-level binary recompilation system, where the specialization capability is limited [8].

3.2.2 Function Specialization

As noted in (Section 3.1.3), reducing the scope of optimization to individual functions reduces the overhead of compilation. When specialization is applied

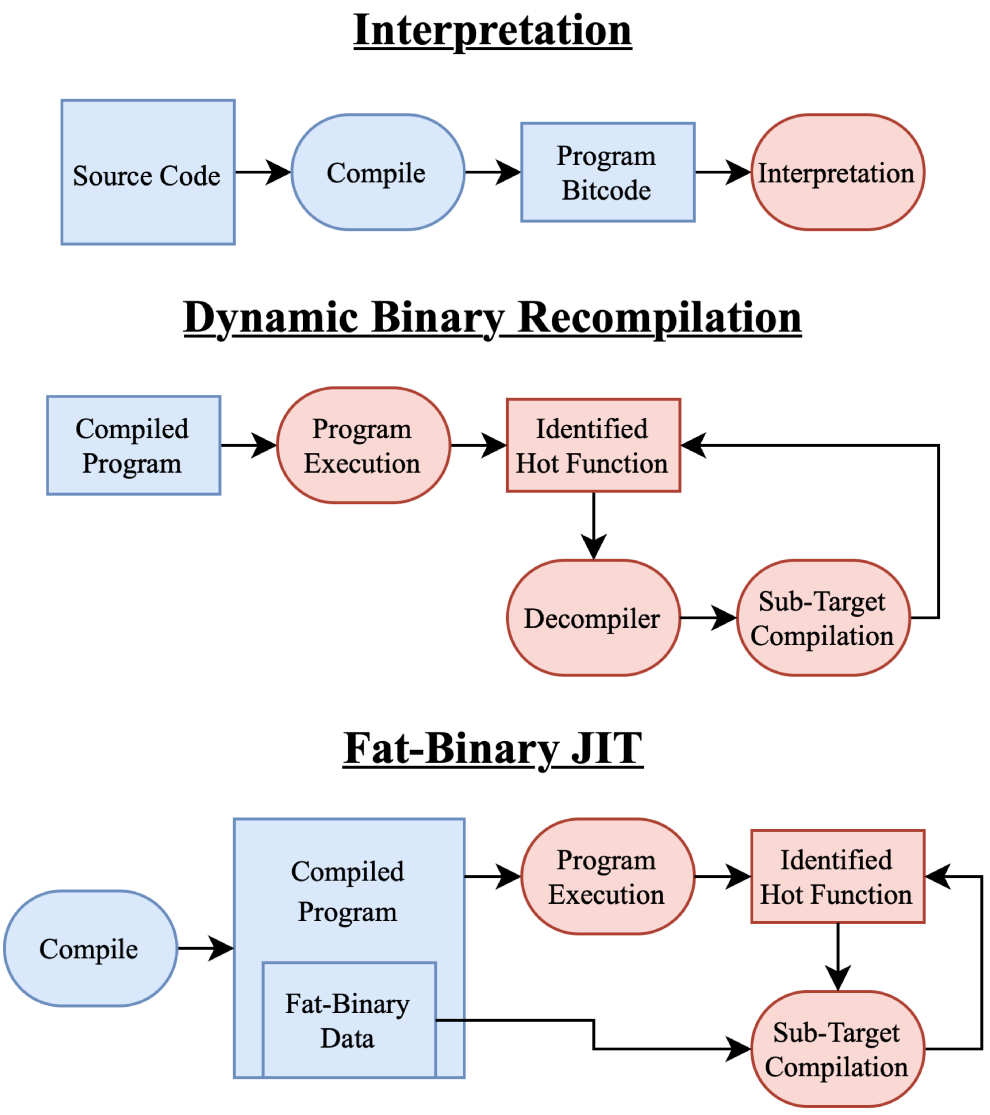

Figure 3.2: Different methods of dynamic sub-target specialization. Stages occurring before program execution are shown in Blue and those occurring during are in Red.

dynamically, this takes greater importance, as it reduce execution time overhead. Beyond reduction, function specialization differs in how the code for optimization is retrieved, of which we note two key approaches below.

Binary Recompilation

Binary recompilation consists in raising the level of representation of the program from its binary form to an intermediate representation level that can be used by an optimizing compiler. The ability to raise the representation is limited and the overhead of recompiling an entire application can be high. Therefore, a solution is to reduce the scope and limit the binary recompilation to functions of interest where performance gains are expected. This reduction of scope is performed by dynamic recompilation methods to reduce the overhead of recompilation and focus the optimization improvements where it is important [16]. Alternatively, targeted recompilation may be used to insert instrumentation, debug information, or code changes into a compiled binary [15]. The downside of this approach is that the time taken to decompile the function code is an additional overhead, and the IR raised from the binary can lack information, limiting the recompiler's capability to specialize to a sub-target.

Fat Binary JIT

The process of creating the IR is costly and the ability of binary analysis to reconstruct the IR is limited. Furthermore, the IR reconstruction replicates work already performed during the AOT compilation. Thus, instead of recreating IR, the system could preserve the IR originally constructed from the source code. This approach is referred to as a **fat binary** [40], wherein IR and AOT compiled code are packed together, allowing dynamic compilation systems to access and compile the preserved IR. By doing so, this method removes the overhead of decompilation. When using a fat-binary JIT system for target specialization, the selection of which portions of the program should have their IR preserved to enable recompilation is important because optimizations that are unlikely to change with specialization should be performed by the AOT

compiler and do not require JIT compilation. Additionally, saving the IR increases the binary file size and may expose proprietary code design that can inform the development of exploits targeting the program. Thus, security concerns may require IR obfuscation which increases both the AOT compilation time and the JIT compilation time.

Chapter 4

Dynamic Adaptive Sub-Target Specialization (DASS)

The core idea of this thesis 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 [40]. Aside from sub-target specialization, DASS also enables applications to benefit from improvements in new releases of a compiler without AOT recompilation. This section presents DASS's core design (Section 4.1) — from code segment cloning to dispatch and execution of sub-target-specialized code at runtime — and a proof-of-concept implementation in LLVM (Section 4.2). Sources of overhead imposed by this prototype implementation and the core concept of DASS (Section 4.3) are identified as potential downsides.

4.1 Core Design

DASS is a compiler system that compiles 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 4.1.1); 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 4.1.2); 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 4.1.3); 4. *Dynamic Sub-target Specialization*: generation of sub-target-specialized code at run-time (Section 4.1.4). Both *Fat Binary Creation* and *Dynamic Redirection Setup* happen at compile-time, while *Symbol Resolution* and *Dynamic Sub-Target Specialization* happen at run-time.

4.1.1 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 CFG 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 4.1.4).

4.1.2 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 code

[1] JITed is a neologism for Just-In-Time compiled.

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.

4.1.3 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.

4.1.4 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.

4.2 DASS Prototype Implementation

The **proof-of-concept** (POC) implementation of DASS is realized inside the LLVM project [29] 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, the LLVM `C/C++` frontend, to support explicitly marking functions to be JITed (Section 4.2.1); 2. an LLVM IR pass that copies the IR and inserts dynamic-dispatch logic into marked functions (Section 4.2.2); 3. a JIT runtime library built on top of LLVM's ORC JIT (Section 4.2.3); 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 4.2.4).

4.2.1 Extensions to Clang

```
int foo(int x) __attribute__((ijit)) {
  return x - 2;
}
```

Listing 4.1: Function annotated with the ijit attribute.

In the POC implementation the `C/C++` function attribute `ijit` (Listing 4.1) marks functions for sub-target specialization through JITing with the DASS runtime. As discussed in Section 4.1, candidate functions can be identified through profiling, static analysis, or run-time cost/benefit analysis. In this POC implementation, to focus experimentation on the potential speedup of selective specialization, no automatic mechanism to select functions is implemented. Instead, two criteria, measuring the fraction of execution time attributed to a function and the static sub-target specialization speedup, are used to apply attributes to the functions of an application (See Section 5.1.2). These criteria approximate the effects of an effective automatic selection method.

4.2.2 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

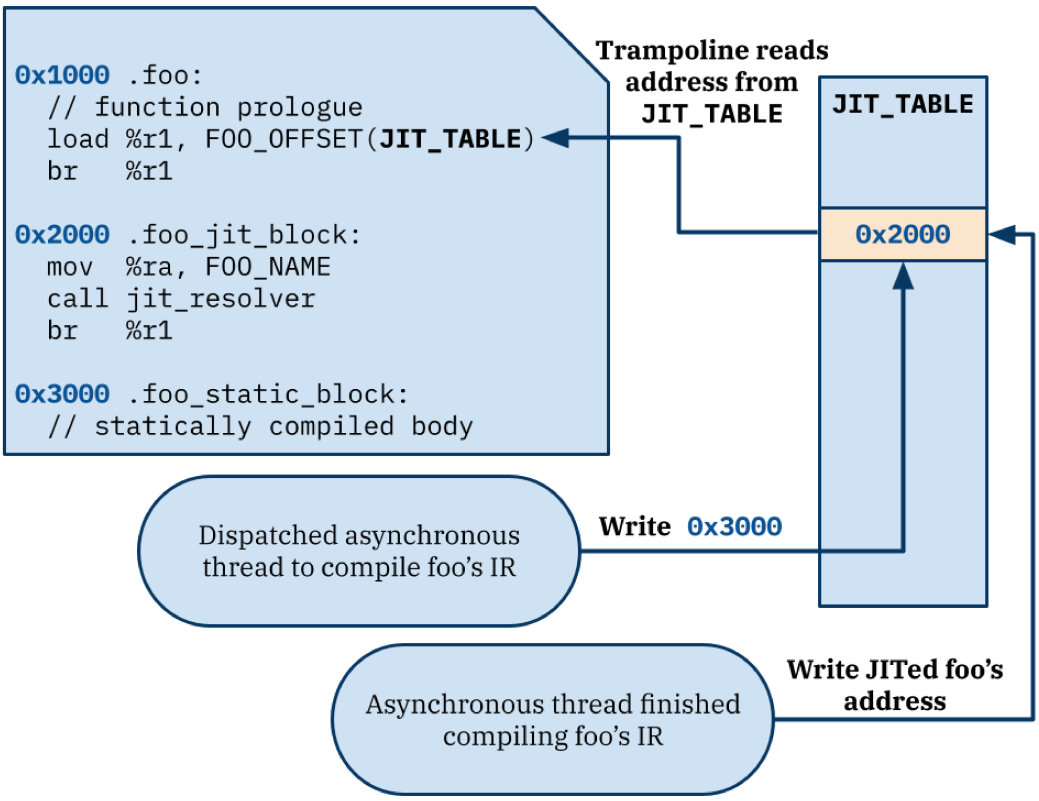

Figure 4.1: Diagram of the JIT call trampoline.

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 during execution, after the IR is JITed (Section 4.2.3). All the information obtained by DASS's IR pass is kept under compiler-reserved variables[2] that are stored in dedicated binary sections (Section 4.2.4).

Once the marked functions are copied, DASS's IR pass inserts a trampoline at the entry point of the copied functions' body. The trampoline mechanism is outlined in Figure 4.1. 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

[2]Not accessible by the application code.

of the statically compiled function body. While asynchronous compilation completes, all calls to the function execute this body. Once the IR string is compiled the table is atomically updated, thus future trampoline loads will jump to the JITed code.

In the POC implementation, functions are copied immediately 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 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 (Chapter 5). Therefore, not performing AOT compiler transformations that depend on sub-target information is just a policy to avoid premature specialization.

4.2.3 JIT Runtime

DASS's runtime is built on top of **On Request Compilation** (ORC) v2 API [31]. ORC is a library for building JIT compilers. Languages such as `Lua` [22], `C++` [56], and `Swift` [60] 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 4.2.2). 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.

4.2.4 Extensions to LLD

DASS requires 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 4.2.2) 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.

Table 4.1: Sources of specialization overhead with the DASS prototype.

Overhead	Source
Compilation	Fat-Binary IR Size Fat-Binary IR Features Optimization Pipeline
Initialization	Fat-Binary IR Size
Indirection	Calls to JITed Functions Calls to AOT-compiled Symbols in JITed Code
Optimization	AOT-compiled Symbols stored in Prototype Structures
Locality	AOT-compiled Code Distance in Memory from JITed Code

4.3 DASS Overhead

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 (Chapter 5), are summarized in Table 4.1. Details on each are listed and discussed below.

4.3.1 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. Furthermore,

asynchronous compilation still requires compute resources, preventing their usage in other computation. For the applications used in this thesis' experimental evaluation (Section 5.4.1), compilation time was not a significant source of overhead — up to 0.46% of an application's execution time (See Table 5.5).

4.3.2 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 5.4.1, 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.

4.3.3 Indirection Overhead

This overhead comes from the trampoline structure used as a dynamic-dispatch mechanism (Section 4.1.2). The cost of executing the table load and indirect branch is incurred every time a selected segment is executed. For frequently called segments, this indirection overhead can become significant (Section 5.4.2).

A less obvious source of indirection overhead is paid when accessing variables or calling functions in the AOT-compiled binary from the JITed segment. The addresses of variables and functions defined in the AOT-compiled binary are 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 call functions in the AOT-compiled binary (Section 5.5.1). This overhead can be eliminated by using instructions that directly reference the address of such symbols; whether this is possible depends on the code model that is used by the JIT compiler.

4.3.4 Optimization Limitation Overhead

While DASS focuses on optimizing specific code segments, the data structures and code adjustments necessary to facilitate that dynamic specialization can have inadvertent impacts on the optimization and generation of the surrounding AOT-compiled code. The LLVM GlobalOpt pass identifies global variables that never have their address taken. For those instances reads/writes of the variable can be simplified as the compiler can verify when and where the value in the variable is modified. Considering programs with frequent accesses to a global variable, this pass can provide a substantial speedup to total program execution time.

As it relates to DASS, implementation of the prototype requires registering the symbols (global variables and functions) called within a selected segment to the JIT engine. To achieve this, the addresses of global variables must be saved to a data structure that is accessed by the DASS runtime. By accessing these addresses, all global variables used in selected segments cannot be optimized by GlobalOpt both in the JITed code and in all AOT-compiled code. While this is the most obvious and impactful (Section 5.4.3) instance where DASS can induce inadvertent overhead by limiting optimization, other instances can exist and may emerge in future compiler optimizations.

4.3.5 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 segment 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 5.5.1).

Chapter 5

Evaluation

The proof-of-concept implementation of DASS is used to evaluate the feasibility of dynamic sub-target specialization of `C/C++` code through a fat-binary JIT approach. In the following sections, the evaluation answers the questions:
① How the performance of sub-target-specialized code compares to target-specialized code? (Section 5.2.1)
② Can the improvements of sub-target specialization be achieved without recompiling the whole application? (Section 5.2.2)
③ Can DASS achieve the improvements of static sub-target specialization? (Section 5.3)
④ What are the sources of overhead in the current implementation of DASS? (Section 5.4 and Section 5.5)

A summary of the key results from each question is presented in Section 5.6.

5.1 Experimental Setup and Methodology

The evaluation is performed on the `C/C++` benchmarks from the SPEC CPU2017 suite. Benchmarks are compiled with the SPEC *base*[1] 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.

[1]All benchmark modules are compiled with the same set of compiler flags passed in the same order, unless explicitly stated otherwise.

Table 5.1: Machine configurations for the evaluation.

Name	Architecture	Processors	RAM
Machine-9	POWER9 ™	20	492 GB
Machine-10	POWER10 ™	24	895 GB

All experiments are performed on two machines, one with a POWER9 CPU (Machine-9) and the other with a POWER10 CPU (Machine-10), the compute resources for each are shown in Table 5.1. Experimental results are presented as the average of three program executions, unless stated otherwise. Variance between executions was measured and found to be minor throughout all executions, at most 0.265% of total execution time. Most performance measurements are presented as a speedup from the execution time of the benchmark entirely compiled with POWER7 sub-target specialization, which is referred to throughout this chapter as "*P7*". Performance metrics used to measure overhead are collected through hardware counters with the Linux Profiler, `perf`.

5.1.1 DASS Prototypes

Three prototype variants of DASS are evaluated:

- `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.
- `D-Async`: Selected functions are 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 execute the JITed code.
- `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.

A diagram of each prototype's sequence for performing JIT compilation of selected functions is shown in Figure 5.1.

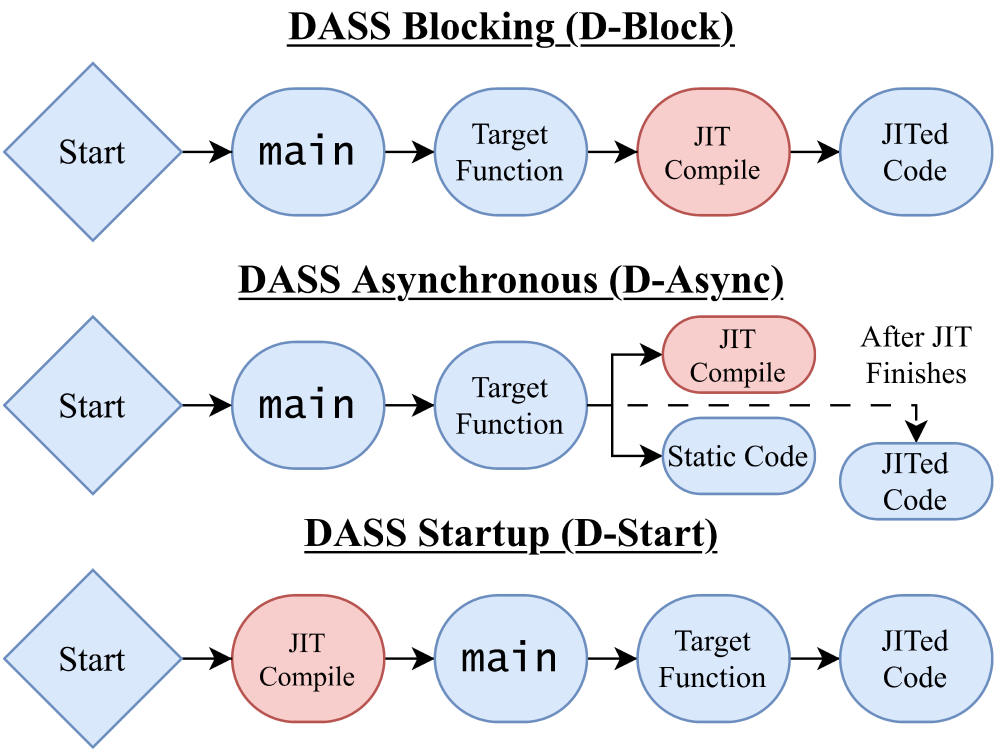

Figure 5.1: Process for performing dynamic compilation of target functions for each DASS prototype.

5.1.2 Specialization & Selection Methods

This evaluation analyzes 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, for Machine-`10`, `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 the executing machine: 1. the number of `perf` samples that hit f represents at least $XX\%$ of total samples in the code compiled for the machine's sub-target; and 2. the number of samples that hit f decreases by $YY\%$ between the profiles obtained by applying whole-program specialization for the `POWER7` sub-target and the machine's sub-target. 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 the executing machine's sub-target, 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 f to the total execution time.

The number of functions, and their corresponding percentage of IR instructions and function execution time relative to the total execution time of a sub-target specialized run, is shown for both machines in Table 5.2 and Table 5.3. Benchmarks wherein no function falls into any selection are omitted. A detailed list of each function selected by the criteria is included in Appendix A.

Template functions were not considered for specialization because of limitations in the DASS prototype. Marking selected functions is performed manually with a `C/C++` attribute in the source code, as such for template functions naively adding the attribute applies specialization to all template instantiations created by the compiler. Furthermore, creating explicit instantiations with the attribute for each selected template function was infeasible for the number of template

Table 5.2: Number of benchmark functions (Fn) and percentage of program IR and execution time (Time) corresponding to functions selected by each criteria for Machine-9.

Benchmark	E10-S10			E05-S05			E02-S02		
	Fn	IR %	Time %	Fn	IR %	Time %	Fn	IR %	Time %
perlbench	0	N/A	N/A	0	N/A	N/A	3	0.49%	7.93%
mcf	0	N/A	N/A	0	N/A	N/A	1	12.9%	23.6%
namd	2	2.08%	28.6%	8	9.97%	77.8%	13	16.5%	99.4%
povray	2	0.17%	28.7%	3	0.20%	36.0%	8	0.81%	50.2%
lbm	1	19.1%	99.1%	1	19.1%	99.1%	1	19.1%	99.1%
omnetpp	0	N/A	N/A	0	N/A	N/A	2	0.05%	20.4%
xalancbmk	0	N/A	N/A	0	N/A	N/A	2	0.11%	10.3%
x264	1	0.15%	24.6%	3	0.27%	41.7%	10	2.49%	63.1%
blender	0	N/A	N/A	0	N/A	N/A	2	0.07%	5.35%
deepsjeng	0	N/A	N/A	2	10.1%	15.8%	9	24.1%	42.0%
imagick	2	1.42%	66.1%	3	1.43%	71.8%	4	1.48%	93.1%
leela	0	N/A	N/A	3	0.98%	20.6%	10	4.19%	52.2%
nab	1	3.75%	59.3%	2	4.69%	68.4%	4	5.32%	99.0%
xz	0	N/A	N/A	0	N/A	N/A	2	0.95%	50.0%

Table 5.3: Number of benchmark functions (Fn) and percentage of program IR and execution time (Time) corresponding to functions selected by each criteria for Machine-10.

Benchmark	E10-S10			E05-S05			E02-S02		
	Fn	IR %	Time %	Fn	IR %	Time %	Fn	IR %	Time %
perlbench	0	N/A	N/A	1	0.26%	7.81%	5	0.93%	18.5%
namd	2	2.10%	29.0%	8	10.0%	72.7%	13	16.5%	99.5%
povray	1	0.09%	14.4%	4	0.26%	49.4%	8	0.81%	62.3%
lbm	1	19.1%	99.5%	1	19.1%	99.5%	1	19.1%	99.5%
omnetpp	0	N/A	N/A	0	N/A	N/A	6	0.25%	38.5%
xalancbmk	0	N/A	N/A	0	N/A	N/A	1	0.003%	2.17%
x264	2	0.24%	39.6%	3	0.27%	46.2%	11	2.58%	69.1%
blender	0	N/A	N/A	0	N/A	N/A	1	0.01%	2.58%
deepsjeng	0	N/A	N/A	2	10.1%	13.9%	10	25.1%	38.1%
imagick	3	1.48%	87.2%	3	1.48%	87.2%	3	1.48%	87.2%
leela	0	N/A	N/A	3	0.69%	21.0%	6	1.82%	33.8%
nab	1	0.29%	11.5%	3	4.38%	76.7%	4	5.32%	86.5%
xz	0	N/A	N/A	0	N/A	N/A	2	16.0%	10.7%

functions in ***parest***. Performing selection as a compilation pass would remove this limitation, as such template functions pose no challenge for the core concept of dynamic specialization. Library functions were also ignored. Libraries do not include their source code within their binary, restricting recompilation, and replacing calls to the library functions, with alternative code that is inserted during AOT compilation, is not possible with the prototype.

These restrictions do not greatly limit function selection, there is only one instance of a library function, in ***nab***, which has 6.4% of total program samples. Ignoring template functions disallows ***parest***, apart from that, only one function in ***leela*** cannot be selected, which has 2.3% of total program samples.

5.2 Static Specialization

This section examines the program speedup from static specialization by comparing the execution time of benchmarks compiled for newer sub-targets, up to the executing machine's sub-target (`POWER9`/`POWER10`), with respect to *P7*. Comparing the speedup achieved by performing sub-target specialization, both across the whole program (Section 5.2.1) and only on select functions (Section 5.2.2), answers questions ① and ②.

5.2.1 Whole-Program Specialization

Every experiment is performed independently for both machines outlined in Table 5.1, afterwards similarities and differences are identified and explained.

Machine-`9` Results

Figure 5.2 presents the speedup achieved with whole-program sub-target specialization on Machine-`9`. Speedup varies between 85% for ***x264***, to a minor slowdown in ***omnetpp*** of 1.6%. Seven benchmarks saw at least 10% speedup over *P7* when compiled for the machine sub-target (`POWER9`), encouraging the conclusion that specialization achieves substantial performance gains.

Analyzing code differences between the same benchmark compiled with *P7*

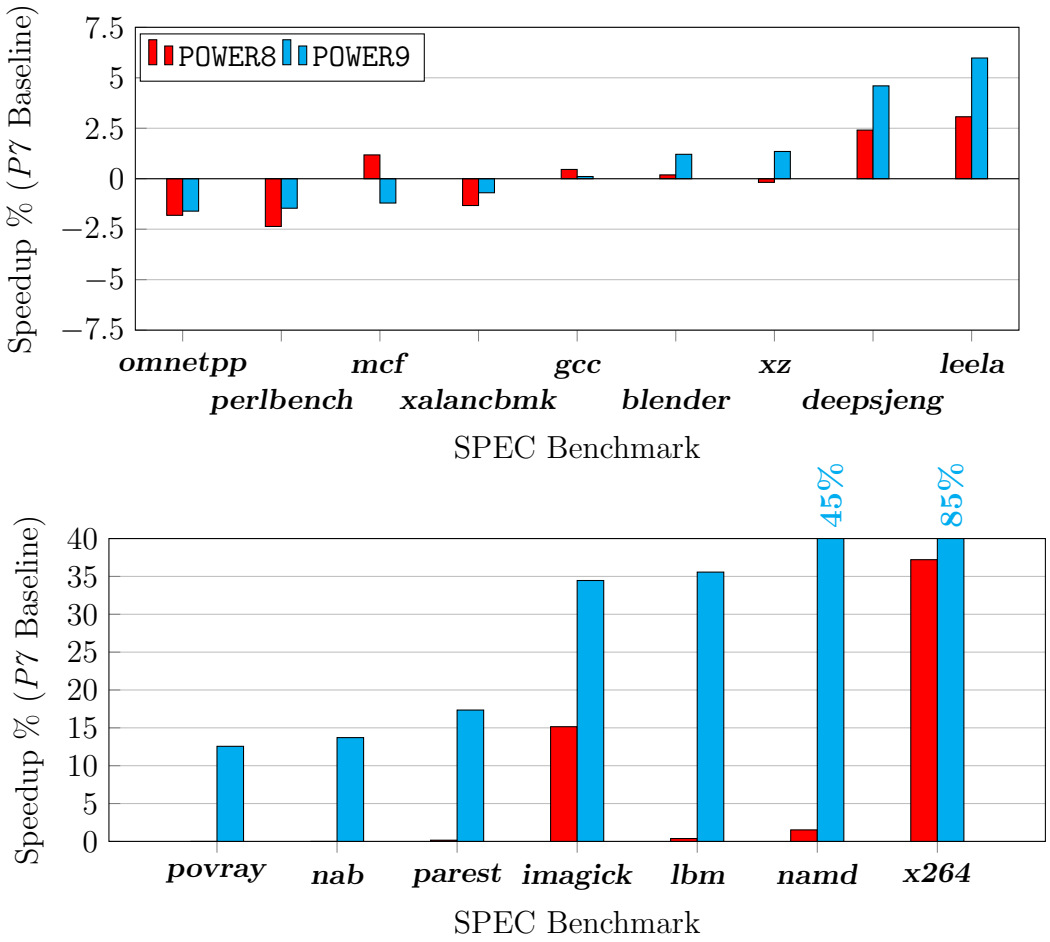

Figure 5.2: Sub-target specialization speedup percentage for a given sub-target, in respect to *P7*, on Machine-9. Ordered by POWER9 speedup.

and the POWER9 sub-target reveals the source of specialization speedup and identifies benchmark's with code that benefits from new processor hardware instructions. The significant program speedup 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. ***namd*** sees substantial speedup, caused by the POWER9 compiler using instructions such as **extswsli** to improve execution time performance. ***lbm*** and ***imagick*** display substantial POWER9 speedup, both from POWER9 targeted IR transformations that reduces the size of the binary code, and through instruction scheduling optimizations that improve data locality and reduce memory stalls on POWER9.

Multiple benchmarks display minor instances of slowdown when specialized to POWER9. Differences in compiler priorities, which achieve speedup when applied to benchmarks like ***namd*** have adverse impacts on function code with different features, as is the case with the slowdown measured for ***omnetpp***. POWER9 targeted code generation replaces the **sldi** instruction with **extswsli** in the function **cSimulation::selectNextModule**. The computation performed by the instructions is effectively covered by **sldi**, while **extswsli** performs a more costly operation (sign-extension) that causes the slowdown. Despite these instances of performing higher cost computation, in terms of program execution time, these instances represent at most a slowdown of 1.6% with POWER9 specialization, while the speedup from specialization on other benchmarks is far greater.

Machine-10 Results

Figure 5.3 presents the speedup achieved by whole-program sub-target specialization on Machine-10. Specialization speedup varies from a 218% speedup in ***x264*** to a slowdown in ***mcf***, with a greater range of speedup found then on Machine-9. The speedup for ***x264*** identifies the advantage of specialization and this benefit is further shown with significant speedup from ***lbm*** and ***imagick***. However, the greater variance of speedup reflects that the advantage

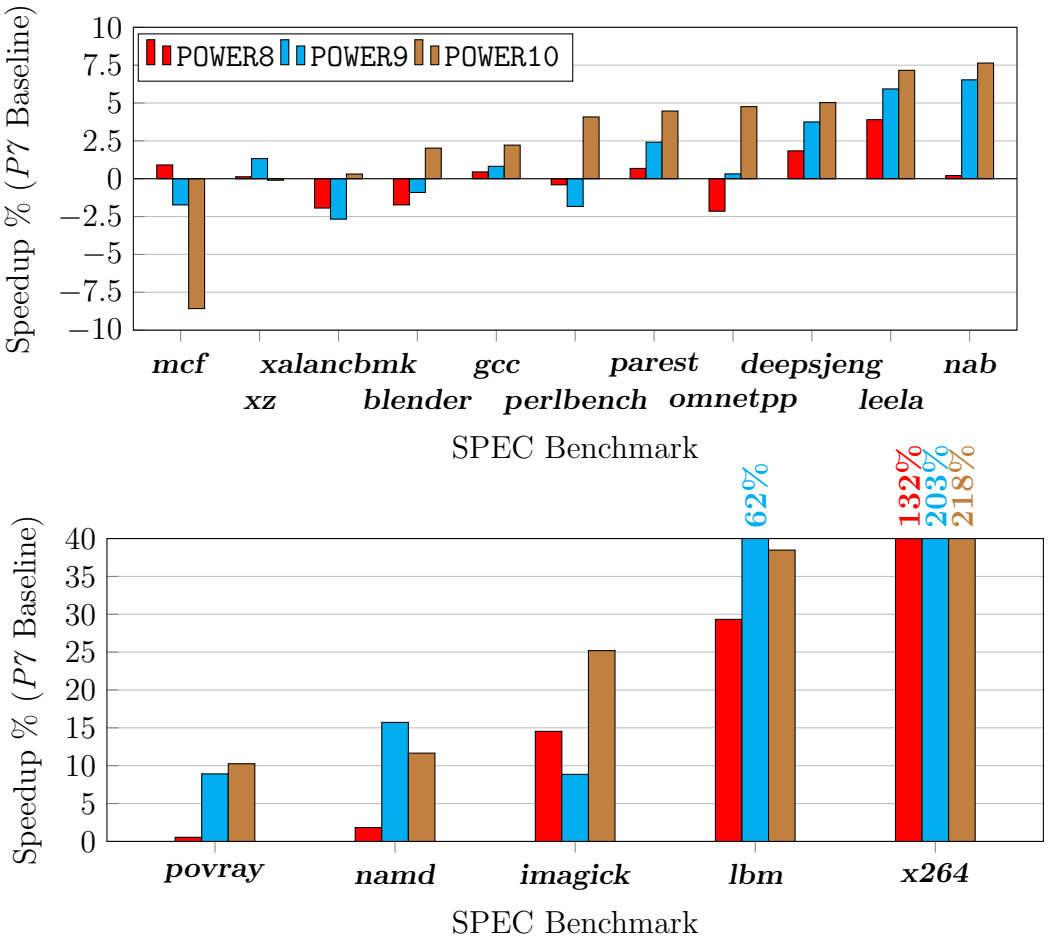

Figure 5.3: Sub-target specialization speedup percentage for a given sub-target, in respect to *P7*, on Machine-10. Ordered by POWER10 speedup.

of sub-target specialization depends on the features of the program; on how the compiler generates code; and on the availability of applicable hardware instructions.

For instance, the significant improvement in ***x264*** comes from instructions mentioned in Section 5.2.1, alongside further `POWER10` optimization. For the function **`x264_pixel_var_16x16`**, the execution time is reduced by 6x through introducing the vectorization intrinsic **`lvm.vector.reduce.add`** to the function IR and the corresponding vector instructions during code generation. Improvement from `POWER10` specialization is also seen with ***imagick***, where the prefixed instruction **`paddi`**, new to `POWER10`, is used to replace multiple **`addi`** instructions in hot code.

All evaluation is performed with a compiler version that is still in development where some of the profitability analyses are still being deployed, which 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.

Conclusions

The results in this section indicate that sub-target-specialized code achieves significantly higher speedups than target-specialize code, providing an answer to ①. Across both machines, significant execution time speedup can be achieved by optimizing program code to the current hardware sub-target. 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 (Machine-`10`) or even slowdown (Machine-`9`). These results encourage limiting the scope of optimization to code that might produce significant speedup, ignoring functions that do not benefit from specialization.

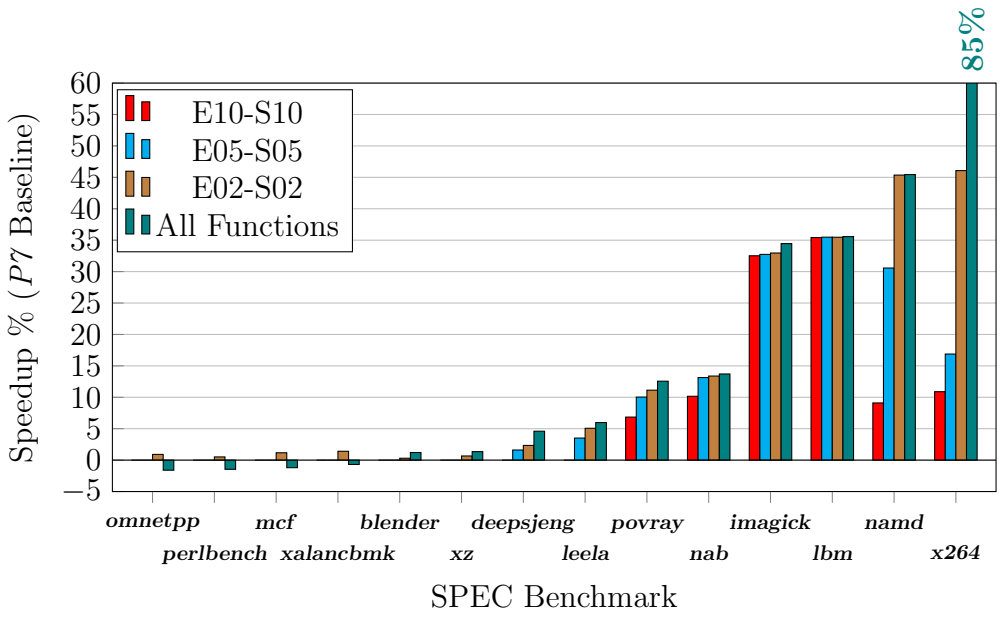

Figure 5.4: Speedup percentage attained by different function specialization criteria for `POWER9`, relative to *P7*, on Machine-9.

5.2.2 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 5.1.2, are specialized. Specialization is performed statically; the rest of the program is compiled for `POWER7`. Effective instances of specialization are identified by a close speedup to that reached by whole-program sub-target specialization. Further explanation of results is informed by the percentage of whole-program `POWER9`/`POWER10` program execution time, shown in Table 5.2 and Table 5.3, that is accounted for by the criteria selected functions. A key assumption is that programs with consistent specialization across benchmark code should attain roughly the same percentage of whole-program specialization speedup as their percentage of program execution time. Instances where specialization speedup is greater indicates code that has more specialization potential within the program. When selecting functions for DASS, greater specialization potential is preferable.

Machine-9 Results

Figure 5.4 compares the speedup from only specializing functions selected by each criteria to POWER9, with whole-program specialization for POWER9. Specializing only criteria selected functions is able to attain a significant portion of total specialization speedup, especially for E02-S02. This conclusion is supported when considering the percentage of execution time accounted for by selected functions, shown in Table 5.2.

When the execution time dominated by the criteria selection is close to the total program execution time ($> 90\%$), the proportion of full POWER9 specialization attained by selective specialization corresponds to the execution time dominance. For instance, ***namd*** with E02-S02 covers 99% of execution time, and attains 99% of POWER9 specialization speedup. Similar behaviour, is shown for ***imagick***, where E02-S02 covers 93% of execution time, and attains 96% of POWER9 speedup. This is expected, as speedup is only attainable from improving functions that are a significant portion of runtime. Furthermore in instances were almost all execution time is covered by criteria functions, almost all attainable specialization speedup should be achieved.

However, when criteria selection execution time dominance is a smaller portion of the total ($< 90\%$), more variable behaviour with regards to specialization speedup proportion is shown. For instance, ***x264*** at E05-S05, covers 42% of execution time, but attains only 20% speedup. Analysis of whole-program specialization of ***x264*** compiled for POWER7 and POWER9 reveals that the total program samples decreases by 42% when function profiling is performed. While the sample reduction of the three E05-S05 functions: **x264_pixel_satd_8x4**, **get_ref** and **mc_chroma**, is 20%, 36% and 31% respectively. From this it is evident that the lower selective function speedup emerges from the criteria selecting functions that speedup less than the benchmark average function speedup. Looking at the profiles of other ***x264*** functions finds: **x264_pixel_sad_16x16** and **x264_pixel_sad_x4_8x8** with sample reductions of 84% and 88% respectively. Both dominate substantial program execution time for POWER7 (16% and 7.7%), however their POWER9 execution time dominance is decreased so

much that they are missed by the E05-S05 criteria (reduced to 4% and 1.6%). These results indicate a limitation of the selection criteria, with the selection focusing on recompilation of functions that have a substantial execution time on the executing hardware, it sometimes misses opportunities to recompile functions that once specialized will no longer be dominant in the total program execution time. This encourages future work to consider how hot functions will vary by sub-target specialization, and discourages assumptions of uniformity in terms of what functions dominate execution time between different sub-targets.

Compared to the sub-optimal selection for ***x264***, other benchmark criteria selections attain speedup significantly higher than the combined execution time dominance of the selected functions. For ***imagick*** at E10-S10, a 66% combined execution time dominance attains 94% of whole-program `POWER9` specialization speedup. This is caused by the function **`MorphologyApply`**, which is selected by the criteria and sees a 36% reduction of samples when compiled to `POWER9`, compared to the average reduction of 26% for the whole benchmark. Similar behaviour is seen with ***povray*** at E02-S02, with a combined execution time dominance of 50% that attains 89% of whole-program whole-program specialization speedup. This comes from greater specialization sample reduction from the functions **`All_Sphere_Intersections`** (24% reduction), **`Inside_Plane`** (34% reduction), and **`Inside_Quadric`** (36% Reduction), compared to the whole-program specialization sample reduction from `POWER7` to `POWER9` of 12%.

Machine-`10` Results

Figure 5.5 compares the speedup from only specializing functions selected by each criteria to `POWER10`, with whole-program specialization for `POWER10`. For certain benchmarks selective specialization attains close to whole-program specialization while targeting only a few functions. 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*** has notably low criteria selection specialization speedup. The gap between execution time dominance (46%) and percentage of

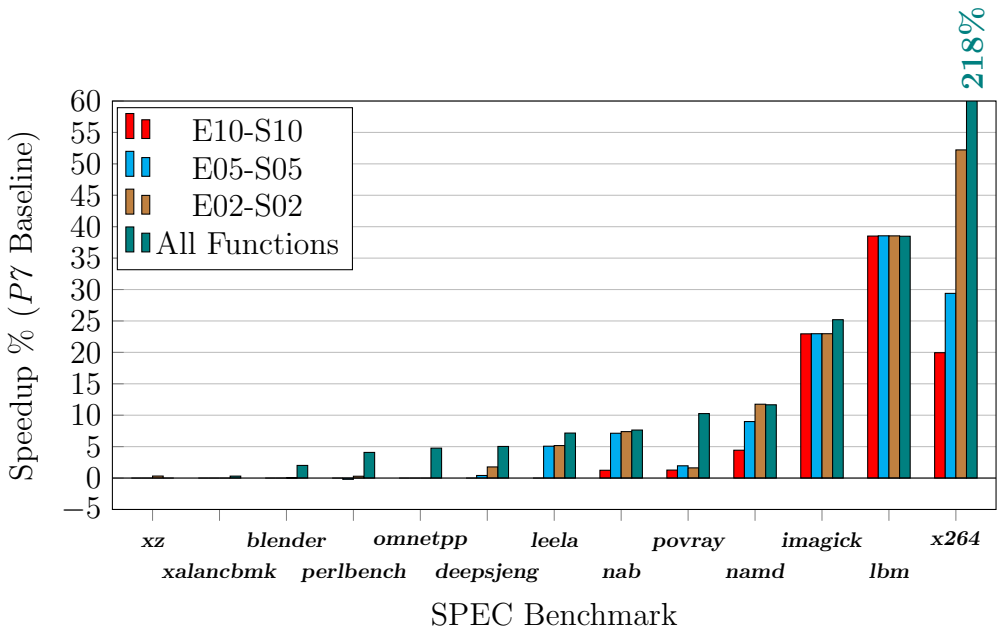

Figure 5.5: Speedup percentage attained by different function specialization criteria for `POWER10`, relative to *P7*, on Machine-`10`.

whole-program `POWER10` specialization speedup (14%) is significant. The cause of this gap comes from not including the function **`x264_pixel_sad_x4_8x8`**, that sees a 97% reduction in samples from `POWER7` to `POWER10`, in the function selection criteria. The reason this function is missed is because from `POWER7` to `POWER10` it moves from having the largest execution time percentage (16%) in terms of profile samples, to only 1.3% of samples when the benchmark is compiled to `POWER10`, due to specialization speedup. In comparison, **`x264_pixel_satd_8x4`** is selected by the criteria, despite a specialization sample reduction of 14% compared to the average sample reduction of 62%.

For most benchmarks, increasing the number of functions selected for specialization decreases the performance gap relative to whole-program specialization. ***povray*** is the only exception where speedup decreases from E05-S05 to E02-S02. Analysis reveals that the generated code for whole-program specialization has fewer `nop` instructions than the code generated with criteria selection 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%.

Conclusions

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. This is encouraged by the results from ***lbm***, though with the caveat that vendor programs face varied workloads and are unlikely to see a program with as dominant a function and as clear an optimization opportunity. Furthermore, the opportunity for sub-target specialization speedup is not equally distributed in program functions, as shown with ***imagick*** on Machine-`9` and ***namd*** on Machine-`10`. Those instances further encourage reducing specialization scope, as careful selection can attain near whole-program specialization speedup, while performing specialization on functions that take a moderate proportion of total program execution time.

This evaluation identifies complications with attaining sub-target specialization through selecting hot functions. The lower speedup present with ***x264*** on both machines emerges from selecting functions with significant execution time when specialized for the machine sub-target, ignoring the execution time when compiled generically. The difference of which functions are dominate in program execution time between generic and specialized compilation complicates selection. Knowledge of a programs hot functions for a given machine cannot accurately reflect the ideal functions to select for specialization, as generic compilation can exhibit significantly different function execution time dominance. Furthermore, function selection for this evaluation is performed with benchmark profiles formed by combining the results from each executed workload. Real world scenarios exhibit greater variation in workloads then any benchmark suite and for most programs attaining full awareness of how

different workloads impact function hotness is very difficult. As such, the task of identifying appropriate functions for specialization is complex and necessitates substantial future study.

5.3 DASS Specialization

The conclusions from Section 5.2 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 5.2.2) to the dynamic specialization of the same functions with 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. Furthermore, limitations in the current prototype implementation make execution of many of the functions in the larger E02-S02 criteria fail.

One benchmark — ***namd*** — is excluded from both machine comparisons because limitations in the DASS prototype prevent it from specializing functions for the program. ***namd*** defines some hot functions (*e.g.* pairlist_from_pairlist) in headers such that multiple definitions of the function exist across different program files. When read from the fat-binary by the DASS runtime, these definitions overwrite eachother and break the JIT engine. For Machine-`10`, ***nab*** is excluded as the JIT engine is restricted to a small code model which when applied to the selected function code for ***nab*** results in segmentation faults that prevent benchmark execution. In both instances, benchmark execution is limited by the current prototype, and the limitation is not inherent to the core ideas and design of DASS.

Machine-`9` Results

Figure 5.6 compares the speedup when functions selected by the criteria are statically specialized to `POWER9`, to the speedup when DASS performs the specialization of the select functions dynamically. Comparing static and dynamic specialization, even with the best performing prototype variant `D-Block`, the

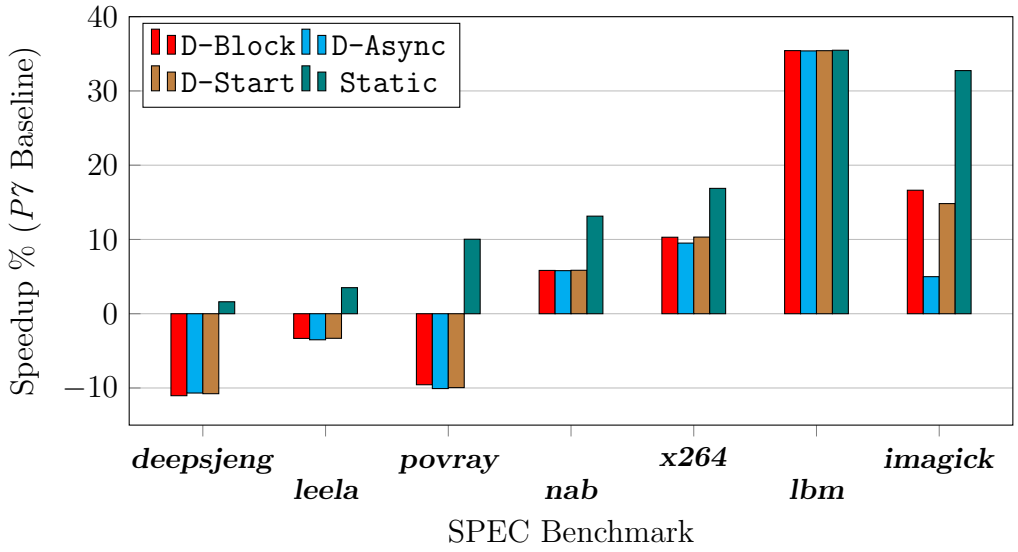

Figure 5.6: Speedup percentage of static specialization and DASS prototypes for `POWER9` with the E05-S05 selection criteria, on Machine-9.

specialization speedup of DASS diverges significantly with the static speedup for most benchmarks. Only ***lbm*** achieves roughly equal performance to static specialization, while ***nab***, ***x264***, and ***imagick*** have significantly lower DASS speedup than static specialization, with a speedup reduction of 56%, 39% and 49% respectively. Furthermore, `D-Block` causes slowdown from *P7* for ***deepsjeng***, ***leela***, and ***povray***. These results are especially poor as the benchmarks perform better when compiled generically than if dynamic sub-target specialization is performed. Explanation of these results encourages an analysis of what additional costs are imposed by the DASS prototype compared to static specialization.

Comparing the results of DASS prototypes, significant variation in speedup for the same benchmark occurs in some instances. While `D-Block` and `D-Start` exhibit similar speedup for all benchmark runs, `D-Async` shows lower speedup than the other prototypes for ***imagick***. For `D-Async`, execution of static AOT-compiled generic code until asynchronous compilation completes, results in a 70% reduction in speedup compared to `D-Block`. Analysis of the selected functions reveals that two of the three selected functions, **`MeanShiftImage`** and **`MorphologyApply`**, have only one and two calls for the entire bench-

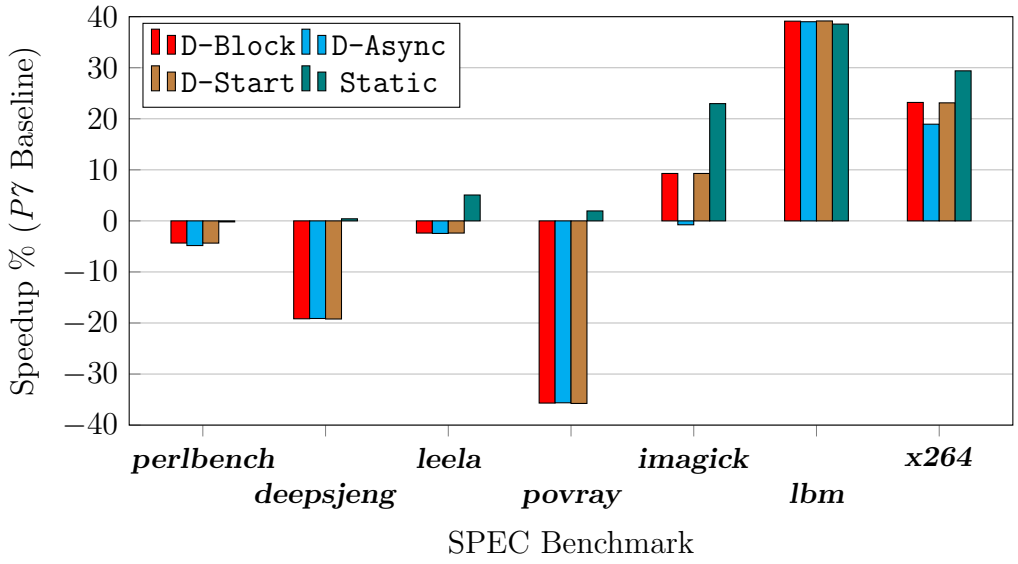

Figure 5.7: Speedup percentage of static specialization and DASS prototypes for `POWER10` with the E05-S05 selection criteria, on Machine-`10`.

mark execution respectively. This low call count, combined with `D-Async` executing at least one call of any function with AOT-compiled generic code while compilation completes, is the cause of the observed speedup reduction. For **`MorphologyApply`**, 50% of function execution is performed with AOT-compiled code, while for **`MeanShiftImage`**, all function execution is performed with the generic code. This generic code execution, alongside the results of the JIT compilation of **`MeanShiftImage`** never being executed, is the cause of the lower speedup seen between `D-Async` and `D-Block`/`D-Start`.

Machine-`10` Results

Figure 5.7 compares the speedup when the functions selected by the criteria are statically specialized to `POWER10`, compared to speedup when DASS performs specialization of select functions dynamically. Mixed results, in line with those for Machine-`9` are found when comparing static and dynamic specialization. `D-Block` is the best performing prototype variant, and ***lbm*** performs closest to static speedup. While, ***imagick*** has significantly lower speedup when specialization is performed dynamically (60% reduction), the gap of static vs dynamic speedup for ***x264*** decreases from a 39% reduction on Machine-`9`, to a

21% reduction on Machine-10. Finally, ***perlbench***, ***deepsjeng***, ***leela***, and ***povray*** slowdown when dynamically specialized with DASS compared to *P7*. Significant slowdown for ***povray*** with -36% speedup, encourages a study of potential overhead sources that DASS can introduce.

Comparison of speedup between prototype variants shows significant variance between `D-Block`/`D-Start` and `D-Async` for the benchmark ***imagick***. The reduction in speedup from `D-Block` to `D-Async` is 108% for the benchmark. Slowdown comes from **`MeanShiftImage`**, which exhibits greater execution time dominance compared to benchmark execution on Machine-9 (41% vs. 31%). As the function runs exclusively in the AOT-compiled generic code when `D-Async` is measured (because of a singular function call over the entire benchmark execution), the speedup reduction increases on Machine-10 from Machine-9.

Conclusions

Experimental results from both machines indicate that `D-Block` and `D-Start` show little difference in speedup for all tested benchmarks. In applications with multiple workloads where selected functions are not called in all workloads, blocking could improve over startup because extraneous compilations would be avoided. However, for the analyzed benchmarks all functions are called on each workload, minimizing the difference between the two approaches. In comparison, the asynchronous compilation variant, `D-Async`, has noticeable slowdown when performed on ***imagick*** for both machines. Two factors contribute to this slowdown. First, while the functions are being compiled the non-sub-target-specialized AOT-compiled code is used and, as the results in Section 5.2.1 show, the performance difference between target and sub-target-specialized code can be significant. Second, 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. Furthermore, for the benchmarks used in this evaluation, the advantage of asynchronous compilation is limited because their compilation time is insignificant, as shown in Section 5.4.1.

Speedup comparisons between static specialization and the DASS prototype

variants from both machines indicate an affirmative but dependent answer to ③. Considering the best-performing variant `D-Block`, dynamic specialization through DASS is only able to match static specialization speedup for certain benchmarks that exhibit highly desirable characteristics for dynamic function compilation. Variation in dynamic specialization effectiveness can 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 code generated by the DASS prototype and by the AOT compiler points to further differences. Whether these additional differences are inherent to the idea of DASS or are a side-effect of the prototype design is important to a proper evaluation of the concept. Thus, in the spirit of always measuring one level deeper [41], an in-depth study of the various sources of overhead for DASS is necessary.

5.4 DASS Overhead

As discussed in Section 4.3, there are five distinct sources of overhead in the current DASS implementation: 1. Compilation Overhead (Section 4.3.1): the cost of performing sub-target specialization at run-time; 2. JIT Runtime Initialization Overhead (Section 4.3.2): the cost of reading fat-binary data and setting up the ORC and LLVM's components for JITing; 3. Indirection Overhead (Section 4.3.3): introduced by the trampoline structure used as a dynamic-dispatch mechanism that enables the AOT-compiled code to call JITed functions; 4. Code Optimization Limitations (Section 4.3.4): limitations to code optimization applied both to surrounding AOT-compiled code and to the selected function code; and 5. Lower Code & Data Locality Overhead (Section 4.3.5): 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. Analysis of the first four overheads is performed for both test machines in the following sections; analysis of locality overhead is performed in Section 5.5.

All overhead is calculated using `D-Block` and compared to the total execution time of benchmarks that are run with the prototype. `D-Block` is analyzed as it performs the simplest version of JIT compilation, such that program startup (`D-Start`) and adjacent computation (`D-Async`) are not impacted. This avoids confounding factors that can impact compilation time measurement.

5.4.1 Initialization & Compilation Overhead

Measurement of the overhead for initializing the DASS runtime and performing JIT compilation of functions during execution was performed for all benchmarks, and compared to the total benchmark runtime when executed with `D-Block`. Compared to other instances of overhead both initialization and compilation are one-time costs, that occur once per the entire benchmark execution (initialization) or once per target function for each benchmark execution (compilation). Because of this one-time cost, the longer a given benchmark executes, the less this overhead should impact total execution time. Despite this amortization, both initialization and compilation are affected by the amount of IR that has been saved in a fat-binary for JIT compilation. Throughout this section the number of IR instructions saved to the fat-binary for a function or group of selected functions is referred to by the general term lines-of-code (LOC). Furthermore, the JIT compilation time of a function influences whether asynchronous compilation is valuable, as longer JIT compilation that blocks execution can noticeably delay important program execution.

Machine-`9` Results

For Machine-`9`, measured overhead of both initialization and compilation shown in Table 5.4 is comparatively low as a proportion of program execution time. Analysis of the raw cycle count for initialization and compilation for each benchmark provides insight into how these one-time overhead costs change based on function selection and code behaviour.

Initialization overhead exhibits a correlation between LOC and the number of cycles required for initialization. Creation of the JIT engine instance acts as the constant factor in initialization overhead, while processing the fat-binary

Table 5.4: Percent of total `D-Block` compiled benchmark cycles for initialization of the DASS runtime and JIT compilation of the selected functions from the E05-S05 selection criteria for Machine-`9`. LOC is the number of IR lines.

Benchmark	# Fn	LOC	DASS Initialization		JIT Compilation	
			Cycles	% Total	Cycles	% Total
leela	3	342	2031K	0.0005%	49M	0.0121%
povray	3	387	2600K	0.0004%	28M	0.0046%
lbm	1	505	2051K	0.0019%	40M	0.0362%
x264	3	664	6820K	0.0026%	540M	0.2060%
deepsjeng	2	1579	4655K	0.0019%	145M	0.0578%
nab	2	1709	5323K	0.0021%	188M	0.0727%
imagick	3	4549	13235K	0.0040%	1115M	0.3385%

IR and associated symbols is the source of this correlation. This pattern holds for most instances, however it is broken for ***x264***, wherein a relatively low LOC produces a higher initialization overhead than expected (albeit still insignificant). This variance comes from the execution behaviour of ***x264***, as it is the only benchmark run on Machine-`9` that invokes the program binary three separate times during its workload. As such, the cost of initialization is payed three times, causing the disproportionate overhead. ***lbm*** displays the opposite behaviour, with a lower overhead than ***povray*** despite a higher LOC. For this instance, the cause is symbols, as ***lbm*** has none, while ***povray*** must register ten symbols at initialization. Furthermore, ***povray*** registers three functions, while ***lbm*** registers one, each of which has their own data structures that are read separately.

Compilation overhead displays a similar correlation. LOC and the compilation time grow with each-other for most benchmarks, ***x264*** (three invocations tripling JIT compilation) and ***leela*** disrupt this trend. ***leela*** has a higher raw compilation time than ***lbm***, despite a lower LOC. Differences in the number of functions or symbols are not the answer, as ***leela*** has an equal number of functions and less symbols (3 to 10) than ***povray***, despite having a longer compilation time.

Analysis of ***leela***'s compilation overhead by function reveals that **`self_atari`**

Table 5.5: Percent of total `D-Block` compiled benchmark cycles for initialization of the DASS runtime and JIT compilation of the selected functions from the E05-S05 selection criteria for Machine-`10`. LOC is the number of IR lines.

Benchmark	# Fn	LOC	DASS Initialization		JIT Compilation	
			Cycles	% Total	Cycles	% Total
leela	3	240	1358K	0.0005%	18M	0.0068%
povray	4	500	2360K	0.0006%	32M	0.0074%
lbm	1	505	1635K	0.0020%	30M	0.0353%
x264	3	664	5211K	0.0027%	362M	0.1837%
perlbench	1	1149	7628K	0.0035%	193M	0.0879%
deepsjeng	2	1579	3279K	0.0018%	101M	0.0545%
imagick	3	4705	9277K	0.0058%	732M	0.4564%

comprises 55% of the total JITed IR, but incurs 66% of the compilation overhead. Comparing statically optimized IR reveals the function has a constant-bound loop which is unrolled when compiled for `POWER9`. This operation differentiates **`self_atari`**, from the other selected functions for ***leela***, which do not significantly modify the code through IR transformations. Unrolling increases the number of IR instructions, which extends the compilation time for the given function, causing ***leela*** to have a higher compilation overhead.

Machine-`10` Results

Overhead results from Machine-`10` (Table 5.5) show a low comparative overhead for both initialization and compilation. Correlation between LOC and initialization/compilation is present. Instances of variance come from the aforementioned symbol registration (***povray***), and multiple benchmark invocations (three for ***perlbench*** and ***x264***). Compilation time variation comes from the extent of optimization performed for the given benchmark functions.

Conclusions

Experimental results from both machines exclude initialization and compilation from being significant contributors to the speedup reduction seen for the DASS prototype in Section 5.3. Compared to program execution time, both initialization and compilation time overhead is insignificant for the analyzed

benchmarks. However, considering hypothetical programs with shorter execution time (evidenced by the multiple executions of ***perlbench*** & ***x264***), initialization/compilation overhead can become significant. Thus the amount of IR saved to the fat-binary compared to the execution time of program workloads is an important consideration when applying a DASS based approach to specialization. Furthermore, the results from ***leela*** on Machine-9 reveal that more aggressive compiler transformations can significantly impact JIT compilation time, a potential limitation on the extent of applied sub-target specialization when considering programs with shorter execution times.

5.4.2 Indirection Overhead

To facilitate JIT compilation of selected functions and later redirect execution to JITed code, the DASS prototype implements a JIT trampoline structure (described in Section 4.2.2). To measure the indirection overhead induced by this structure, a modified version of the prototype (called `Indirect`) is contrasted with results from `Static` and `D-Block`. `Indirect` performs the same JIT trampoline indirection as `D-Block`, but does not perform run-time compilation and redirects execution to an AOT-compiled copy of the selected function instead of JITed code. To isolate indirection overhead from other factors, the AOT-compiled function copy is sub-target specialized via the same function attribute used to specialize `Static`. The extent of overhead induced by the DASS trampoline structure is shown by comparing the speedup of `Indirect` to the speedup of `Static` and `D-Block`. Speedup that is closer to `D-Block` than `Static` indicates that the indirection necessary for the DASS prototype is the dominant source of dynamic specialization overhead.

Trampoline overhead comes from the added instructions and function call. Thus, the overhead is assumed to be in relation to the number of calls to the selected function, as each call executes the trampoline. Functions that are called infrequently with respect to the benchmark execution time should exhibit less indirection overhead than those executed frequently. To test this assumption the raw execution time difference between `Indirect` and `Static`, and the number of calls to selected functions, is shown for each benchmark,

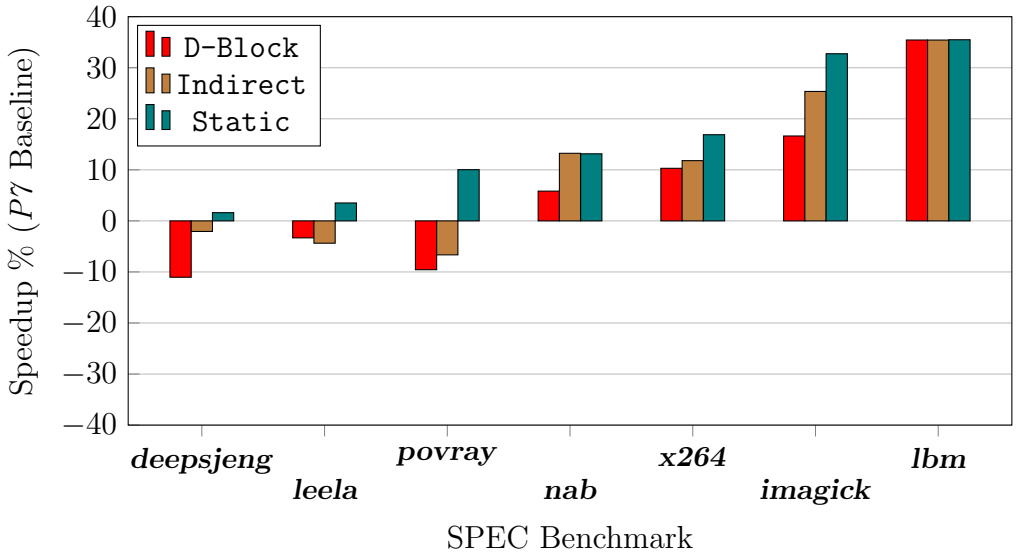

Figure 5.8: Contrasting speedup on Machine-9 for E05-S05 static specialization (`Static`), dynamic specialization (`D-Block`), and a modification of DASS (`Indirect`), that uses the JIT trampoline to call AOT-compiled code.

alongside the speedup comparison.

Another form of indirection overhead comes from accesses to AOT symbols within the JITed code. Separating this overhead from other factors is more complicated, and often is revealed through loss of locality in data accesses. Thus, this overhead is analyzed through the benchmark profile metrics in Section 5.5.

Machine-9 Results

Experimental results for Machine-9 are shown in Figure 5.8. For applications like ***leela***, ***povray***, and ***x264***, indirection represents the majority of dynamic specialization overhead, while for ***deepsjeng***, ***nab*** and ***imagick***, there is a substantial portion of overhead that occurs separate from the indirection mechanism. Analysis of the indirection overhead reveals that in the instance of ***leela***, the JITed code is faster than static code, as the speedup for `D-Block` improves over `Indirect`. The reason the benchmark has a lower speedup than `Static` with `D-Block` is because of indirection overhead.

Table 5.6 shows how indirection overhead grows with the number of selected function calls for Machine-9. These results are in line with the assumption

Table 5.6: Calls made to DASS E05-S05 selected functions compared to the difference of execution time between `Indirect` and `Static`, for Machine-`9`.

Benchmark	Selected Function Calls	`Indirect` - `Static`	`Indirect` / `Static`
nab	251	-0.438s	-0.09%
lbm	3000	0.091s	0.04%
deepsjeng	1147891208	16.148s	3.71%
x264	2935327434	21.965s	4.55%
imagick	7934662915	42.340s	7.49%
leela	9159189545	59.358s	8.04%
povray	21354156842	172.484s	17.35%

Table 5.7: Calls made to DASS E05-S05 selected functions compared to the difference of execution time between `Indirect` and `Static`, for Machine-`10`.

Benchmark	Selected Function Calls	`Indirect` - `Static`	`Indirect` / `Static`
lbm	3000	-0.312s	0.19%
deepsjeng	1147891208	14.297s	4.72%
x264	3052760393	14.881s	4.06%
perlbench	3453544769	13.838s	3.35%
imagick	7988222453	21.235s	7.63%
leela	11310453124	35.267s	7.31%
povray	23927201310	86.924s	14.43%

that indirection overhead grows with the number of functions calls. Overhead is minimal for benchmarks where the selected functions are called infrequently (***lbm***, ***nab***). While, the overhead both as a proportion of total execution time, and as a raw time difference increases alongside the number of function calls. The slight speedup for ***nab*** with `Indirect` over `Static` comes from variability in benchmark execution time, and is not indicative of `Indirect` improving execution time for certain instances.

Machine-`10` Results

Speedup comparison for `Indirect` on Machine-`10` is presented in Figure 5.9. Similar behaviour to Machine-`9` is present, certain applications are dominated by indirection overhead while other benchmarks demonstrate significant non-indirection overhead (***deepsjeng*** and ***povray***).

The growth of indirection overhead with the number of calls to selected

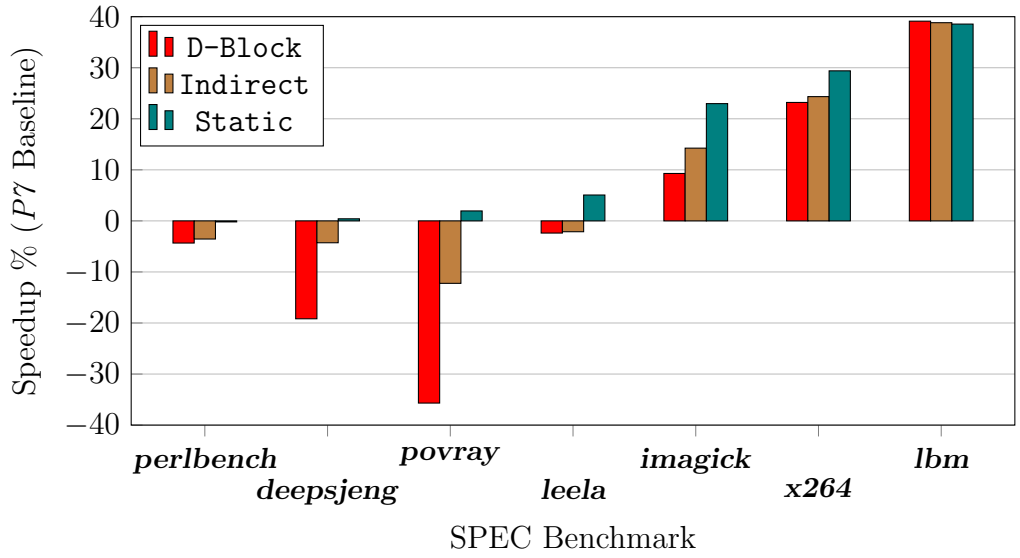

Figure 5.9: Contrasting speedup on Machine-10 for E05-S05 static specialization (`Static`), dynamic specialization (`D-Block`), and a modification of DASS (`Indirect`), that uses the JIT trampoline to call AOT-compiled code.

functions is shown in Table 5.7. The assumption, that a benchmark's raw indirection overhead corresponds to selected function calls, holds for most applications, however ***perlbench*** breaks this assumption, with a lower raw overhead than ***deepsjeng*** but 3x the number of selected function calls. Furthermore, while raw overhead grows with selected function calls for most benchmarks, variation in total benchmark execution time causes the percentage of indirection overhead to decrease with call growth for some benchmark pairs (***imagick*** and ***leela***).

Analysis of performance metrics from `Static` and `Indirect` for ***perlbench*** and ***deepsjeng*** reveals a 24x growth in I-Cache misses for ***deepsjeng*** from `Static` to `Indirect`. This compares to a 1.4x increase in the metric for ***perlbench***, with the significant difference of metric growth being the main source of the relatively high indirection overhead for ***deepsjeng*** compared to ***perlbench*** and ***x264***. The cause of this significant increase in I-Cache misses relates to the creation of a function clone that is called through indirection for each selected function call. This clone is located at a different memory address, as such cloned instructions are not always in the I-Cache when the original function is called. Furthermore, the indirection must access instructions from

both the original function addresses and the cloned function address every call, and these instructions can overlap and evict each other within the cache. This eviction can incur significant slowdown as memory accesses are expensive to perform, explaining the greater overhead of ***deepsjeng***. Similar behaviour can occur with the DASS prototype, as the JITed code instructions are located separately from the AOT-compiled function. This loss of instruction locality is investigated further in Section 5.5.

Conclusions

Evaluation of indirection overhead for Machine-`9` and Machine-`10` identifies that the majority of speedup reduction from static specialization for the DASS prototype is caused by the indirection overhead added by the JIT trampoline structure. Furthermore, correlation between the raw indirection time overhead and the number of calls to functions with the JIT trampoline is present. These results discourage dynamic specialization of short functions with frequent calls, where indirection overhead can overpower the execution time speedup from sub-target specialization. If specialization of short functions is desired, alternatives with lower indirection overhead, such as modification of function call sites (binary rewriting), should be considered, if feasible. Few benchmarks demonstrate substantial overhead separate from indirection (***imagick***, ***deepsjeng***, ***nab***, ***povray***), investigation of these overheads requires deeper analysis.

5.4.3 Optimization Limitation Overhead

To perform JIT compilation and redirection to JITed code the DASS prototype must create additional data structures to register global variables and fat-binary IR. These structures and how they access and use global variables and other data has implications for the optimization of both the surrounding AOT-compiled code, and the IR that is saved for JIT compilation. Optimization limitations incurred by these structures limits the total speedup achievable by dynamic specialization. The overhead incurred by these structures is shown by a comparison of `Indirect` and a further modification that adds the structures but continues to not perform JIT compilation (`Indirect+Struct`).

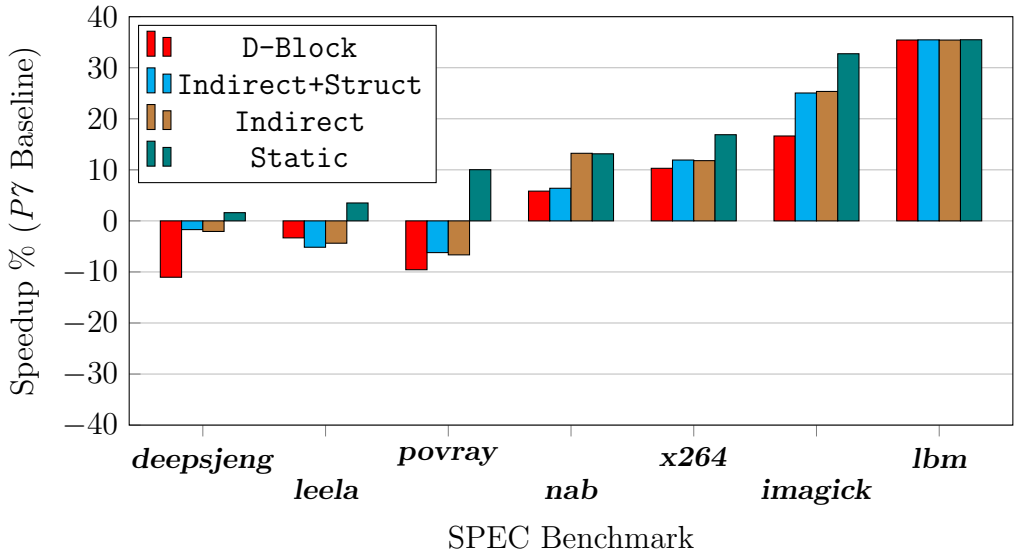

Figure 5.10: Contrasting speedup on Machine-9 for E05-S05 static specialization (`Static`), dynamic specialization (DASS), and two modifications of DASS: `Indirect`, that uses the JIT trampoline to call AOT-compiled code; and `Indirect+Struct`, that includes creation of the JIT data structures to `Indirect`.

Machine-9 Results

Optimization overhead is not a significant factor for most benchmarks when executing on Machine-9 (Figure 5.10), except for the benchmark ***nab***. For ***nab***, significant optimization is applied with static specialization through the LLVM GlobalOpt pass. The JIT structures explicitly access the address of every global variable in the function, preventing these optimizations. When the GlobalOpt pass is disabled for both `Static` and `D-Block`, the two versions achieve a close speedup over *P7* of 11.8% and 11.2% respectively.

Machine-10 Results

Experimental results from Machine-10 (Figure 5.10) show that AOT-compiled code is unaffected by the JIT structures for any of the evaluated benchmarks. ***nab*** fails to execute DASS and as such, the potential overhead incurred to the benchmark from GlobalOpt pass optimization limitations cannot be measured.

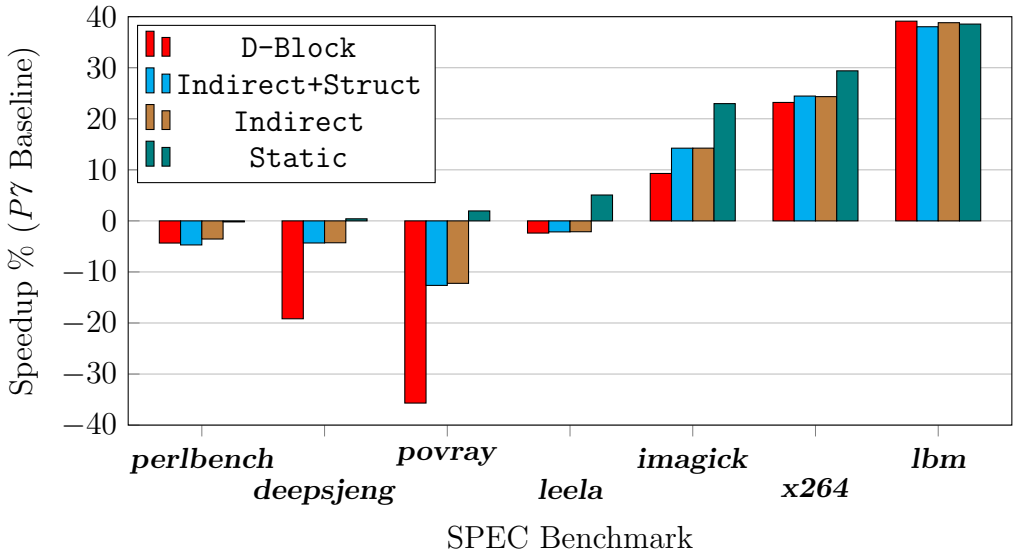

Figure 5.11: Contrasting speedup on Machine-10 for E05-S05 static specialization (`Static`), dynamic specialization (DASS), and two modifications of DASS: `Indirect`, that uses the JIT trampoline to call AOT-compiled code; and `Indirect+Struct`, that includes creation of the JIT data structures to `Indirect`.

Conclusions

Optimization overhead is not a significant factor for most benchmarks measured in this evaluation. However, ***nab*** indicates that the JIT structures, and specifically their access of global variable addresses can incur significant overhead. As such, modifications to the compilation pipeline ordering, to move JIT setup after any limited passes, should be considered. Though, this is not always possible, as moving fat binary creation too far can transform the IR with generic specialization decisions that limit dynamic specialization gains.

5.5 DASS Metric Analysis

Analysis of how overhead impacts JITed code execution, as opposed to the surrounding JIT structures (*e.g.* trampoline), is complicated as speedup of the function body alone can be impacted by timing instrumentation. Thus, to explore this overhead, various metrics are collected from benchmark execution, for both `Static` and `D-Block`. These metrics are shown in Table 5.8 for

Machine-9, and Table 5.9 for Machine-10. Significant differences in metrics between the Static and D-Block versions indicate impacts from the DASS prototype.

Table 5.8: Benchmark metrics comparing `Static` and `D-Block` specialization for E05-S05 on Machine-9. `D-Block`/ `Static` is calculated using profile cycle counts. Normalized metrics where DASS differs from the `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	***deepsjeng***		***leela***		***povray***		***nab***		***x264***		***lbm***		***imagick***	
`D-Block / Static`	10.32%		7.76%		19.41%		6.90%		6.71%		0.04%		13.26%	
	`Static`	`D-Block`	`Static`	`D-Block`	`Static`	`D-Block`	`Static`	`D-Block`	`Static`	`D-Block`	`Static`	`D-Block`	`Static`	`D-Block`
Cycles	1259B	10.32%	2131B	7.76%	2904B	19.41%	1365B	6.90%	1492B	6.71%	621B	0.04%	1632B	13.26%
Instructions	2170B	10.39%	2892B	14.22%	4764B	23.79%	1790B	7.84%	2745B	7.93%	1004B	0.03%	3186B	17.93%
Instruction Count Normalized Metrics (X / Instructions)														
Branch Miss	$1.5e^{-3}$	$1.6e^{-3}$	$3.3e^{-3}$	$2.9e^{-3}$	$1.1e^{-3}$	$1.0e^{-3}$	$4.4e^{-4}$	$4.0e^{-4}$	$3.4e^{-4}$	$4.3e^{-4}$	$1.8e^{-5}$	$1.9e^{-5}$	$3.0e^{-4}$	$2.6e^{-4}$
Branch Load Miss	$1.5e^{-3}$	$1.6e^{-3}$	$3.3e^{-3}$	$2.9e^{-3}$	$1.1e^{-3}$	$1.0e^{-3}$	$4.4e^{-4}$	$4.0e^{-4}$	$3.4e^{-4}$	$4.3e^{-4}$	$1.7e^{-5}$	$1.9e^{-5}$	$3.0e^{-4}$	$2.6e^{-4}$
iCache Miss	$1.9e^{-3}$	$1.5e^{-3}$	$1.1e^{-4}$	$1.7e^{-4}$	$1.9e^{-3}$	$1.8e^{-3}$	$2.2e^{-6}$	$1.0e^{-5}$	$9.8e^{-4}$	$1.1e^{-3}$	$6.4e^{-6}$	$1.0e^{-5}$	$1.2e^{-6}$	$1.9e^{-5}$
dTLB Miss	$1.6e^{-8}$	$1.7e^{-8}$	$1.1e^{-7}$	$9.0e^{-8}$	$2.4e^{-8}$	$2.3e^{-8}$	$4.9e^{-8}$	$3.5e^{-8}$	$5.4e^{-8}$	$5.3e^{-8}$	$2.0e^{-8}$	$1.2e^{-8}$	$1.9e^{-9}$	$6.3e^{-9}$
iTLB Miss	$6.0e^{-7}$	$7.3e^{-4}$	$6.3e^{-7}$	$3.8e^{-7}$	$1.1e^{-6}$	$8.8e^{-6}$	$1.6e^{-7}$	$2.6e^{-8}$	$1.5e^{-6}$	$1.2e^{-6}$	$5.4e^{-9}$	$1.4e^{-8}$	$5.6e^{-9}$	$4.8e^{-8}$
Cycle Count Normalized Metrics (X / Cycles)														
L1 Miss Stalls	0.021	0.021	0.017	0.017	0.035	0.034	$1.4e^{-3}$	$1.3e^{-3}$	$8.6e^{-3}$	$8.4e^{-3}$	$3.3e^{-5}$	$5.8e^{-5}$	$2.3e^{-3}$	$2.4e^{-3}$
FE Stall	0.050	0.046	0.119	0.113	0.039	0.040	0.020	0.019	0.013	0.016	$8.1e^{-4}$	$8.9e^{-4}$	0.020	0.038
BE Stall	0.153	0.158	0.142	0.142	0.163	0.171	0.156	0.153	0.134	0.133	0.148	0.148	0.146	0.133
DERAT Miss Stall	$1.3e^{-3}$	$1.4e^{-3}$	$7.7e^{-4}$	$9.1e^{-4}$	$2.1e^{-4}$	$1.3e^{-4}$	$6.8e^{-6}$	$9.4e^{-6}$	$9.2e^{-5}$	$1.0e^{-4}$	$1.1e^{-5}$	$1.5e^{-5}$	$7.8e^{-5}$	$4.6e^{-4}$
DERAT Miss	$1.3e^{-4}$	$1.3e^{-4}$	$6.9e^{-5}$	$8.9e^{-5}$	$3.4e^{-6}$	$2.1e^{-5}$	$1.1e^{-6}$	$1.7e^{-6}$	$1.7e^{-5}$	$2.7e^{-5}$	$2.5e^{-6}$	$4.8e^{-6}$	$2.3e^{-6}$	$7.1e^{-6}$

Table 5.9: Benchmark metrics comparing `Static` and `D-Block` specialization for E05-S05 on Machine-10. `D-Block`/ `Static` is calculated using profile cycle counts. Normalized metrics where DASS differs from the `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`	5.16%		12.62%		8.06%		37.34%		10.07%		-0.36%		4.10%	
	`Static`	`D-Block`	`Static`	`D-Block`	`Static`	`D-Block`	`Static`	`D-Block`	`Static`	`D-Block`	`Static`	`D-Block`	`Static`	`D-Block`
Cycles	1602B	5.16%	1180B	12.62%	1875B	8.06%	2335B	37.34%	1085B	10.07%	639B	-0.36%	1574B	4.10%
Instructions	3814B	5.79%	2170B	10.32%	2873B	14.28%	4714B	32.16%	3201B	15.37%	1055B	0%	2756B	8.05%
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	0.033	0.035	0.028	0.040	0.049	0.045	0.018	0.048	0.011	0.012	$5.6e^{-4}$	$6.9e^{-4}$	$9.4e^{-3}$	0.010
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	$1.4e^{-6}$	$1.4e^{-6}$	$1.2e^{-5}$	$7.3e^{-5}$	$1.6e^{-5}$	$1.3e^{-5}$	$1.1e^{-7}$	$9.6e^{-8}$	$1.0e^{-6}$	$9.3e^{-7}$	$1.1e^{-4}$	$1.0e^{-4}$	$5.1e^{-6}$	$5.4e^{-6}$

5.5.1 Lower Code & Data Locality Overhead

The sources of poor locality in the 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. Moreover, placing the JITed function in a different memory location may have an effect on the prediction of existing branches — *e.g.* it may introduce branch aliasing. These potential overhead sources are explored for both machines using the gathered metrics.

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. Therefore, the JITed code accesses symbols in the AOT-compiled program through a table — generated by the JIT runtime (Section 4.2.3). As a result, every access in the JITed code to symbols in the AOT-compiled code goes through an indirection. The indirection adds overhead in terms of more instructions per access but, more importantly, because the table is placed on different memory pages than either the JITed and AOT-compiled code, the indirection may add more overhead in terms of address translation.

Machine-`9` Results

Metrics from Table 5.8 helps explain the speedup reduction for ***deepsjeng*** and ***imagick***, which both have substantial non-indirection overhead. High iTLB miss rates with `D-Block` compared to `Static` indicate poor locality when transitioning from AOT code to JITed code.

For ***imagick***, higher DERAT[2] miss induced stalls reflect higher costs in accessing AOT-compiled symbols within JITed code.

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

Machine-10 Results

The results in Table 5.9 indicate that locality is the source of overhead 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 can explain the 34% increase in branch load misses in ***povray***.

Analyzing the locality impacts from AOT-compiled symbol accesses indicates that: 1. there are more stalls due to DERAT 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.

Conclusions

Analysis of code metrics supports the conclusion that locality overhead, both as a function of code instructions being separated in memory for the AOT-compiled and JITed code, and as an indirection to access AOT symbols in JITed code, explains the remaining memory overhead for the experimental benchmarks. The instance of ***povray*** highlights the issue of executing JITed code that is distant in memory from the rest of the program, as it suffers from significant instruction cache misses. ***imagick***, identifies overhead from symbol access, wherein data cache and DERAT misses become significant. These results encourage future work to reduce the separation in memory space for AOT and JITed code, and to work to reduce the level of indirection for accessing AOT symbols.

5.6 Summary

The feasibility of dynamic sub-target specialization is supported through positive answers to the questions raised in this chapter. Experimental results from LLVM and the DASS prototype are encouraging, but also raise new questions

for future work and considerations that must be made when implementing and applying DASS.

Answering ① affirmatively, results from Section 5.2.1 identify multiple benchmarks where sub-target specialization is a significant optimization (greater than 10% improvement in execution time). Despite this improvement, instances where specialization is minor or even incurs slight slowdown, reveals that specialization as an optimization is not guaranteed value. Instead, it is valuable for instances where program code is amenable to the features unique to sub-targets (*e.g.* ***x264***, ***lbm***).

The conclusions from Section 5.2.2 answer ② by showing certain benchmarks (*e.g.* ***namd***, ***imagick***, ***lbm***) can attain a significant portion of whole-program specialization by statically specializing only a fraction of the total program code. While these results are encouraging, experimentation also reveals an instance where selective specialization has negative impacts on the instruction cache by introducing divergent memory alignment goals between generic and sub-target specialized code. Furthermore, results from ***x264*** reveal that the benefit of specialization varies greatly within a program's functions, indicating that selection of functions for specialization requires careful consideration.

③ is answered with analysis of the DASS prototype, the variant `D-Block` produces positive results for certain benchmarks, ***lbm*** and to a lesser extent ***x264***, wherein dynamic specialization speedup approached that of static specialization. Despite these instances, substantial overhead was observed for many benchmarks, encouraging investigation to answer question ④.

Overhead analysis is a substantial portion of this evaluation, as answering ④ is complicated due to the multiple potential sources of overhead (Section 4.3) and the extent of overhead seen for certain benchmarks (***povray***, ***deepsjeng*** and ***imagick***). Compilation and Initialization overhead are not significant (Section 5.4.1) for the analyzed benchmarks. Despite that, experimentation reveals that the amount of specialized IR correlates with these overheads, and as such consideration must be taken for these costs when executing programs with shorter execution times.

Indirection (Section 5.4.2) is the major cause of overhead for multiple bench-

marks (***perlbench***, ***leela***, ***x264***). Analysis reveals significant overhead can be incurred both by frequent calls to JITed functions, and through the movement of JITed code in memory from the original call site (***deepsjeng***), a factor that is explored further as Locality overhead. For indirection, improvement can be achieved by simplifying the calls to JITed code by rewriting call sites instead of the trampoline structure, however, this is not feasible for many developers.

The DASS prototype faces higher overhead from Optimization Limitation (Section 5.4.3) in the instance of ***nab*** due to the JIT data structures preventing the GlobalOpt pass. This instance encourages further exploration of how DASS is implemented through compiler passes, and if a greater modification of the optimization pipeline is necessary.

The results discussed in Section 5.5.1 indicate that a significant source of 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. Although there is not much that can be done to increase instruction locality, some changes could improve the 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 ABI 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.

5.6.1 Function Selection Takeaways

Evaluation of DASS reveals both instances where function code is amenable to the dynamic specialization method, displaying specialization speedup roughly equivalent to that achieved statically (***lbm***), and code wherein dynamic specialization performs significantly worse than static specialization (***povray***).

Through the evaluation, the relative overheads of these benchmarks was connected with distinct function features and behaviours (IR size, function calls, global symbols). Thus, when considering what functions should be selected manually or automatically for a DASS-based system, specific considerations related to function behaviour and design can be taken.

These selection considerations are listed below:

- **Function Features**: Analysis of specialization speedup in Section 5.2 reveals that instances of significant speedup are explained through the application of new sub-target instructions to the code. Furthermore, the largest instances of speedup relate to utilizing new vectorization capabilities through sub-target instructions. This encourages selecting functions that display features relevant to new sub-target capabilities (*i.e.* vector operations). While it is difficult to identify future sub-target capabilities, analyzing instructions introduced in recent sub-target designs can effectively improve selection.

- **Low IR Proportion**: Data from analyzing compilation time in Section 5.4.1 indicates that the overhead is insignificant for the analyzed functions. Part of these results emerges from the comparatively low amount of IR that was selected and JIT compiled. Applying DASS to real-world programs can require wider selection as hot functions differentiate greatly by workload. With that consideration, the extent of fat-binary IR saved should still be small as, beyond one-time initialization and compilation costs, the trampoline structure necessary to support DASS incurs an overhead on selected functions even if JIT compilation is not performed.

- **Call Frequency**: Indirection overhead was shown to be significant in Section 5.4.2. As mentioned above, DASS incurs an overhead for selected functions through its trampoline structure. With that in mind, functions that are called infrequently relative to their individual call execution time are better suited for DASS selection as the indirection cost occurs once

per call. Furthermore, issues of locality with JITed code (Section 5.5.1) are exacerbated by higher call frequency, as the instruction cache impacts are felt in part as execution moves from AOT-compiled to JITed code.

- **Global Symbol Usage**: Indirection as it relates to accessing global symbols is another noticeable overhead in the evaluation. When considering function selection, if symbol accesses can not be optimized with the JIT compilers code model, it is preferable to avoid functions with frequent global symbol accesses, as each access requires an expensive indirection. Symbols, as they pertain to calls for AOT-compiled functions in JITed code, also impact instruction cache locality, as the distance in memory between the instructions of the two functions is often far greater than in a statically specialized binary.

Chapter 6

Related Work

Related work to DASS focuses on existing methods of performing dynamic compilation with `C/C++` and other statically compiled languages (Section 6.1). Not all dynamic compilation methods are explicitly designed to perform sub-target specialization, however in their implementation many of the steps necessary to perform specialization are supported. Thus, these approaches can be compared to DASS, both in the extent of program code that is dynamically compiled, and in their methodology for identifying compilation targets. Beyond the development of DASS, research has designed and implemented methods of static and dynamic analysis to find program hot functions. Other methods have also been explored for the creation of fat-binaries. These works and others (Section 6.2) are relevant as means of extending and comparing alternatives to DASS with the goal of improving the methods applicability in future work.

6.1 Dynamic Compilation Methods

Dynamic code compilation for static languages has been a persistent point of interest in prior research. For dynamic compilation, there are two major approaches: holistic (Section 6.1.1) and selective (Section 6.1.2). A holistic approach targets the entire program for (re)compilation during execution. In comparison, a selective approach narrows the scope to specific "high-value" segments of the code. While the holistic approach benefits from simplicity in its broad target, selective methods like DASS are subject to additional considerations. A selective dynamic compilation system must have a mechanism

to decide what segments are high-value and worthy of recompilation.

6.1.1 Holistic Dynamic Compilation

Holistic dynamic compilation is expensive compared to a selective approach, as the cost to recompile is significantly greater when targeting the entire program code compared to individual functions. Thus, these methods require more significant optimization improvement to pay-off the cost of compilation. Facing this cost, some holistic methods ignore performance, focusing instead on alternative benefits that justify the cost to performance. Approaches to enable interpretation with `C/C++` commonly follow this path. Cling [56] and Runtime Compiled C++ [9] both focus on enabling rapid application development, trading off high overhead with faster code change feedback when developing smaller applications. Feedback is also the goal of CLIP [33], which is designed to be a simplified teaching tool for `C/C++`. DASS, unlike these methods, focuses on improving program execution time and adopts a selective dynamic compilation approach to reduce recompilation overhead.

For holistic methods, that focus on optimization, the overhead of full program compilation can be reduced through various methods. Dynamo [8] reduces interpretation overhead by recording and compiling short instruction traces into "fragments", with the goal of quickly moving from the costly process of interpreting instructions, to the majority of execution occurring through optimized fragments. While, the work of Hildenbrand [21] reduces the overhead of binary recompilation through the development of a lower-level IR that preserves more of the original compilation work for the program. In contrast, DASS with a selective focus starts out with the equivalent of the "fragments" in Dynamo. Furthermore, the fat-binary design of DASS saves the original compilation IR, thus a lower-level representation is not needed, as the cost of decompiling is avoided, and LLVM IR can be more aggressively optimized.

Holistic methods that seek to improve program performance often utilize a combination of architectural and program behavior information. 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 [2] distributes a compilation process similar to BOLT's, but it also reduces the, potentially significant, overhead incurred by profiling. Lightning BOLT [43] develops a similar process to reduce compilation costs by enabling parallelism within the most costly optimization. It transforms the approach of BOLT 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.

6.1.2 Selective Dynamic Compilation

The choice of which segments to target for specialization influences the possible optimizations that dynamic compilation can perform and the corresponding cost. DyC [20] focuses on individual variables, recompiling the code blocks where these variables appear to generate binaries where these variables can be treated as run-time constants. ADAPT [58] identifies loop nests that lack function calls or I/O, creating multiple experimental versions of the code and picking the best-performing version. Azure [63] 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 [40] and ExanaDBT [50] 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 [40] 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 uses a JIT compiler as a convenient way to produce a prototype,

as DASS can be realized without a JIT.

The identification of recompilation targets needs to balance the amount of information required for the decision and the overhead of retrieving that information and making the decision. DyC [20] and Tick C [17] makes use of a manually applied annotation attached to each code line that identifies it for dynamic compilation. Forgoing manual identification, Calpa [35] extends DyC by applying the system's annotations automatically, requiring a profile of the program to be built in a previous run of the program to inform its decisions. Different than 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.

6.2 DASS Expansion Works

The DASS prototype identifies target functions through function attributes manually applied to source code. While this is sufficient for investigating the feasibility of dynamic specialization, identification and application of attributes to beneficial functions is often not feasible in real-world programs. Thus, exploration of both static analysis to identify initial targeted functions, and dynamic analysis, to confirm the hotness of targeted functions, is an important consideration for future work.

The work of Wu & Larus [62] outlines a static analysis, built on heuristics, that can estimate basic block execution frequencies, and which can be extrapolated to estimate function call frequencies. VESPA [36] builds on this, using a machine learning model trained with profile data to statically predict the branch frequencies and resulting basic block execution frequencies of a program. These methods can be utilized to roughly estimate program hot functions, which can inform automatic identification of target functions.

Dynamic analysis has improved accuracy, however it must contend with acting as an overhead on program execution time, thus research often focuses on

reducing analysis overhead. The work of Lee et al. [30], combines a preliminary static analysis heuristic alongside a low overhead bytecode count to estimate hot functions for Java JIT compilation. Mußler et al. [38] reduces the overhead of function instrumentation through a static analysis of function features that filters out undesirable candidates. While, Servat et al. [51] proposes coarse grain sampling to buttress the construction of execution traces using instrumentation. Hypothetically, considering the application of dynamic analysis to guide DASS, the aforementioned work to reduce overhead is key to preserve sub-target specialization speedup.

The fat binary method utilized by DASS saves the code of specific functions to be recompiled in an optimized form at runtime. Other approaches to preserving program code have been done within LLVM, often focusing on saving the entire program to enable Link-Time-Optimization (LTO). WLLVM [3] provides support for saving the entire bitcode of a program within its object files, from which the program can be recomposed using the attached utility. GLLVM [1] extends this work, enabling better performance through parallelism of the process, at the cost of somewhat restricted functionality. Beyond these works, the LLVM project directly supports embedding bitcode through command-line arguments. While the DASS prototype design only needs the target function IR saved, expansion of the work can make use of these techniques to enable a greater selection of functions, at execution time, for recompilation.

Chapter 7

Conclusion

This work evaluated the feasibility of a compiler-based system to apply dynamic adaptive sub-target specialization (DASS). Existing methods of specialization were reviewed, including the style that DASS builds from (Fat-Binary JIT). From that review, the core design and a prototype implementation of DASS was introduced. Results obtained through 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 is sufficient to specialize only a fraction of the application code in order to achieve the majority of the whole-application specialization gains. Furthermore, empirical evidence indicates that in some instances it is possible to attain similar results by applying dynamic specialization at run-time using the DASS prototype. 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.

The results of this thesis both encourage the development of dynamic sub-target specialization techniques, and identify future work in the form of limitations and overheads faced by the DASS prototype. While DASS is used to evaluate the cost-benefit of sub-target specialization, the concern of identifying the target functions for this optimization is not covered. Methods that perform automatic identification and marking of target functions through static and/or

dynamic analysis are necessary to see adoption of dynamic sub-target specialization in real-world applications. Beyond identification, evaluation of DASS reveals future work to optimize the prototype implementation. Lower impact structures for initiating and redirecting execution to JITed code could greatly improve the competitiveness of dynamic specialization when compared to static specialization. While, optimization targeted at reducing the loss of locality between JITed and AOT-compiled code are necessary to avoid high overheads for many functions. These improvements and more can enable dynamic sub-target specialization for real-world applications, improving the performance of programs and increasing utilization of modern processor hardware features.

References

[1] I. A. M. et al. "Whole program LLVM in Go." Accessed: 2022-06-09, SRI International. (2022), [Online]. Available: https://github.com/SRI-CSL/gllvm. 69

[2] S. T. et al. "Propeller: Profile guided optimizing large scale LLVM-based relinker." Accessed: 2022-05-13, Google Inc. (2020), [Online]. Available: https://github.com/google/llvm-propeller/blob/bb-clusters/Propeller%5C_RFC.pdf. 67

[3] T. R. et al. "Whole program LLVM." Accessed: 2022-06-09. (2021), [Online]. Available: https://github.com/travitch/whole-program-llvm. 69

[4] K. Anand, M. Smithson, K. Elwazeer, *et al.*, "A compiler-level intermediate representation based binary analysis and rewriting system," in *Proceedings of the 8th ACM European Conference on Computer Systems*, 2013, pp. 295–308. 10

[5] *Arm® Architecture Reference Manual Armv8, for Armv8-A Architecture Profile*, Arm Limited, Jan. 2021. 1

[6] Aycock, John, "A Brief History of Just-In-Time," *ACM Computing Surveys (CSUR)*, vol. 35, no. 2, pp. 97–113, 2003. 5

[7] Backus, John, "The history of Fortran I, II, and III," *ACM Sigplan Notices*, vol. 13, no. 8, pp. 165–180, 1978. 5

[8] V. Bala, E. Duesterwald, and S. Banerjia, "Dynamo: A transparent dynamic optimization system," in *Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, 2000, pp. 1–12. 12, 66

[9] D. Binks, M. Jack, and W. Wilson, "Runtime compiled C++ for rapid AI development," *Game AI Pro: Collected Wisdom of Game AI Professionals*, p. 201, 2013. 66

[10] 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*, ser. ICPE '18, Berlin, Germany: Association for Computing Machinery, 2018, pp. 41–42, ISBN: 9781450356299. DOI: 10.1145/3185768.3185771. [Online]. Available: https://doi.org/10.1145/3185768.3185771. 2

[11] J.-T. Chan and W. Yang, "Advanced obfuscation techniques for java bytecode," *Journal of systems and software*, vol. 71, no. 1-2, pp. 1–10, 2004. 10

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

[13] 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. 1

[14] L. Eisen, J. Ward, H.-W. Tast, *et al.*, "IBM POWER6 accelerators: VMX and DFU," *IBM Journal of Research and Development*, vol. 51, no. 6, pp. 1–21, 2007. 1

[15] A. Engelke and M. Schulz, "Instrew: Leveraging LLVM for high performance dynamic binary instrumentation," in *Proceedings of the 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, 2020, pp. 172–184. 14

[16] A. F. Engelke, "Optimizing performance using dynamic code generation," Ph.D. dissertation, Technische Universität München, 2021. 14

[17] 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*, ser. POPL '96, St. Petersburg Beach, Florida, USA: Association for Computing Machinery, 1996, pp. 131–144, ISBN: 0897917693. DOI: 10.1145/237721.237765. [Online]. Available: https://doi.org/10.1145/237721.237765. 68

[18] 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. 2

[19] A. Gal, B. Eich, M. Shaver, *et al.*, "Trace-based Just-In-Time Type Specialization for Dynamic Languages," *ACM Sigplan Notices*, vol. 44, no. 6, pp. 465–478, 2009. 6

[20] 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. 67, 68

[21] D. Hildenbrand, "An optimized intermediate representation for binary rewriting at runtime," 2019. 66

[22] 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. DOI: 10.1002/(SICI)1097-024X(199606)26:6\%3C635::AID-SPE26\%3E3.0.CO;2-P. 21

[23] *Intel® Architecture Instruction Set Extensions and Future Features programming reference*, Intel Corporation, Feb. 2021. 1

[24] M. Ismail and G. E. Suh, "Quantitative overhead analysis for Python," in *2018 IEEE International Symposium on Workload Characterization (IISWC)*, IEEE, 2018, pp. 36–47. 12

[25] 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. 6

[26] 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. 7

[27] G. Krylov, P. Jelenković, M. Thom, *et al.*, "Ahead-of-Time compilation in Eclipse OMR on example of WebAssembly," in USA: IBM Corp., 2021, pp. 237–243. 1

[28] S. Kumar, Y. Wang, C. Young, *et al.*, "Exploring the limits of concurrency in ML training on Google TPUs," *Proceedings of Machine Learning and Systems*, vol. 3, pp. 81–92, 2021. 1

[29] 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. 3, 18

[30] S.-W. Lee, S.-M. Moon, and S.-M. Kim, "Enhanced hot spot detection heuristics for embedded java just-in-time compilers," *ACM Sigplan Notices*, vol. 43, no. 7, pp. 13–22, 2008. 69

[31] LLVM Developer Group, *Orcv2: Design and implementation*, `https://llvm.org/docs/ORCv2.html`, Accessed: 2023-04-25, 2019. 21

[32] C. Lomont, "Introduction to intel advanced vector extensions," Intel, Tech. Rep., 2011. 1

[33] H. Luoma, E. Lahtinen, and H.-M. Järvinen, "CLIP, a command line interpreter for a subset of C++," in *Proceedings of the Seventh Baltic Sea Conference on Computing Education Research-Volume 88*, 2007, pp. 199–202. 12, 66

[34] 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. 1

[35] 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. DOI: `10.1109/MICRO.2000.898079`. 5, 68

[36] A. A. Moreira, G. Ottoni, and F. M. Quintão Pereira, "Vespa: Static profiling for binary optimization," *Proceedings of the ACM on Programming Languages*, vol. 5, no. OOPSLA, pp. 1–28, 2021. 68

[37] J. E. Moreira, K. Barton, S. Battle, *et al.*, *A matrix math facility for Power ISA(TM) processors*, 2021. arXiv: 2104.03142 [cs.AR]. 1, 7

[38] J. Mußler, D. Lorenz, and F. Wolf, "Reducing the overhead of direct application instrumentation using prior static analysis," in *European Conference on Parallel Processing*, Springer, 2011, pp. 65–76. 69

[39] J. Nickolls and W. J. Dally, "The GPU computing era," *IEEE micro*, vol. 30, no. 2, pp. 56–69, 2010. 1

[40] 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. 9, 14, 16, 67

[41] J. Ousterhout, "Always measure one level deeper," *Commun. ACM*, vol. 61, no. 7, pp. 74–83, Jun. 2018, ISSN: 0001-0782. DOI: 10.1145/3213770. [Online]. Available: https://doi.org/10.1145/3213770. 45

[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. 9, 66

[43] 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*, ser. CC 2021, Virtual, Republic of Korea: Association for Computing Machinery, 2021, pp. 119–130, ISBN: 9781450383257. DOI: 10.1145/3446804.3446843. [Online]. Available: https://doi.org/10.1145/3446804.3446843. 67

[44] A. Patwardhan, R. Patwardhan, and S. Vartak, "Self-contained cross-cutting pipeline software architecture," *arXiv preprint arXiv:1606.07991*, 2016. 9

[45] 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. 5

[46] Ritchie, Dennis M., "The Development of The C Language," *ACM Sigplan Notices*, vol. 28, no. 3, pp. 201–208, 1993. 5

[47] Sammet, Jean E, "The Early History of COBOL," *History of Programming Languages*, vol. 1, pp. 199–243, 1978. 5

[48] K. Sangani, "Computer says no [software compatibility]," *Engineering & Technology*, vol. 6, no. 9, pp. 76–77, 2011. 7

[49] M. F. Sanner *et al.*, "Python: A programming language for software integration and development," *J Mol Graph Model*, vol. 17, no. 1, pp. 57–61, 1999. 12

[50] 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. 67

[51] H. Servat, G. Llort, J. Giménez, and J. Labarta, "Detailed performance analysis using coarse grain sampling," in *Euro-Par 2009–Parallel Processing Workshops: HPPC, HeteroPar, PROPER, ROIA, UNICORE, VHPC, Delft, The Netherlands, August 25-28, 2009, Revised Selected Papers 15*, Springer, 2010, pp. 185–198. 69

[52] 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, Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pp. 142–147. 1

[53] B. Stroustrup, *The C++ Programming Language*, 4th. Boston, Massachusetts, United States: Addison-Wesley Professional, 2013, ISBN: 0321563840. 5

[54] A. Telea and L. Voinea, "A tool for optimizing the build performance of large software code bases," in *2008 12th European Conference on Software Maintenance and Reengineering*, IEEE, 2008, pp. 323–325. 9

[55] M. Thom, G. W. Dueck, K. Kent, and D. Maier, "A survey of Ahead-of-Time technologies in dynamic language environments," in *Center of Advanced Studies Conference*, ser. CASCON '18, Markham, Ontario, Canada: IBM Corp., 2018, pp. 275–281. 5

[56] V. Vasilev, P. Canal, A. Naumann, and P. Russo, "Cling–the new interactive interpreter for ROOT 6," in *Journal of Physics: Conference Series*, IOP Publishing, vol. 396, New York, NY, USA: IOP Publishing, 2012, p. 052 071. 12, 21, 66

[57] B. Venners, "The java virtual machine," *Java and the Java virtual machine: definition, verification, validation*, 1998. 12

[58] 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. 67

[59] 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 (CASES'11)*, ser. CASES '11, Taipei, Taiwan: Association for Computing Machinery, 2011, pp. 15–24, ISBN: 9781450307130. DOI: 10.1145/2038698.2038704. [Online]. Available: https://doi.org/10.1145/2038698.2038704. 1

[60] 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, Emerging Programming Paradigms for Large-Scale Scientific Computing, ISSN: 0167-8191. DOI: https://doi.org/10.1016/j.parco.2011.05.005. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167819111000524. 21

[61] D. Williams-King, H. Kobayashi, K. Williams-King, *et al.*, "Egalito: Layout-agnostic binary recompilation," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2020, pp. 133–147. 9, 10

[62] Y. Wu and J. R. Larus, "Static branch frequency and program profile analysis," in *Proceedings of the 27th annual international symposium on Microarchitecture*, 1994, pp. 1–11. 68

[63] E. Yardimci and M. Franz, "Mostly static program partitioning of binary executables," *ACM Trans. Program. Lang. Syst.*, vol. 31, no. 5, Jul. 2009, ISSN: 0164-0925. DOI: 10.1145/1538917.1538918. [Online]. Available: https://doi.org/10.1145/1538917.1538918. 67

[64] Y. Yu, H. Dayani-Fard, and J. Mylopoulos, "Removing false code dependencies to speedup software build processes," in *Proceedings of the 2003 conference of the Centre for Advanced Studies on Collaborative research*, 2003, pp. 343–352. 9

Appendix A

Function Selection

The functions selected for specialization in Chapter 5 are listed below, organized by the machine and criteria for which they were selected. Further information, on machine and criteria is given in the aforementioned chapter.

Table A.1: Functions selected for Machine-9 E10-S10.

Benchmark	Function	Source File	Line
namd	pairlist_from_pairlist	ComputeNonbondedInl.h	32
namd	calc_pair_energy_fullelect	ComputeNonbondedUtil.h	352
povray	All_Plane_Intersections	planes.cpp	103
povray	All_Sphere_Intersections	spheres.cpp	122
lbm	LBM_performStreamCollideTRT	lbm.c	262
x264	x264_pixel_satd_8x4	common/pixel.c	234
imagick	MorphologyApply	magick/morphology.c	3827
imagick	MeanShiftImage	magick/feature.c	2108
nab	mme34	eff.c	3203

Table A.2: Functions selected for Machine-9 E05-S05.

Benchmark	Function	Source File	Line
namd	pairlist_from_pairlist	ComputeNonbondedInl.h	32
namd	calc_pair_energy_fullelect	ComputeNonbondedUtil.h	352
namd	calc_pair_fullelect	ComputeNonbondedUtil.h	351
namd	calc_pair_energy	ComputeNonbondedUtil.h	350
namd	calc_pair_energy_merge_fullelect	ComputeNonbondedUtil.h	354
namd	calc_pair_merge_fullelect	ComputeNonbondedUtil.h	353
namd	calc_pair	ComputeNonbondedUtil.h	349
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
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	MorphologyApply	magick/morphology.c	3827
imagick	MeanShiftImage	magick/feature.c	2108
imagick	GetOneCacheViewVirtualPixel	magick/cache-view.c	768
leela	FastBoard::kill_or_connect	FastBoard.cpp	1214
leela	FastBoard::self_atari	FastBoard.cpp	1271
leela	FastBoard::is_eye	FastBoard.cpp	805
nab	mme34	eff.c	3203
nab	nbond	eff.c	768

Table A.3: Functions selected for Machine-9 E02-S02.

Benchmark	Function	Source File	Line
perlbench	Perl_pp_padsv	pp_proto.h	184
perlbench	Perl_regexec_flags	regexec.c	2788
perlbench	Perl_pp_match	pp_proto.h	153
mcf	price_out_impl	implicit.c	444
namd	pairlist_from_pairlist	ComputeNonbondedInl.h	32
namd	calc_pair_energy_fullelect	ComputeNonbondedUtil.h	352
namd	calc_pair_fullelect	ComputeNonbondedUtil.h	351
namd	calc_pair_energy	ComputeNonbondedUtil.h	350
namd	calc_pair_energy_merge_fullelect	ComputeNonbondedUtil.h	354
namd	calc_pair_merge_fullelect	ComputeNonbondedUtil.h	353
namd	calc_pair	ComputeNonbondedUtil.h	349
namd	calc_self_energy_fullelect	ComputeNonbondedUtil.h	361
namd	calc_self_energy	ComputeNonbondedUtil.h	359
namd	calc_self_energy_merge_fullelect	ComputeNonbondedUtil.h	363
namd	calc_self_fullelect	ComputeNonbondedUtil.h	359
namd	calc_self_merge_fullelect	ComputeNonbondedUtil.h	362
namd	calc_self	ComputeNonbondedUtil.h	358
povray	All_Plane_Intersections	planes.cpp	103
povray	All_Sphere_Intersections	spheres.cpp	122
povray	Inside_Plane	planes.cpp	226
povray	All_Quadric_Intersections	quadrics.cpp	136
povray	Inside_Quadric	quadrics.cpp	283
povray	DNoise	texture.cpp	548
povray	Intersection	objects.cpp	93
povray	Intersect_Light_Tree	lbuffer.cpp	1279
lbm	LBM_performStreamCollideTRT	lbm.c	262
omnetpp	cMessageHeap::shiftup	simulator/cmessageheap.cc	195
omnetpp	cMessageHeap::insert	simulator/cmessageheap.cc	166
xalancbmk	ValueStore::isDuplicateOf	ValueStore.cpp	218
xalancbmk	XPath::executeMore	XPath.cpp	307
x264	x264_pixel_satd_8x4	common/pixel.c	234
x264	get_ref	common/mc.c	232
x264	mc_chroma	common/mc.c	263
x264	x264_pixel_sad_16x16	common/pixel.c	61
x264	pixel_hadamard_ac	common/pixel.c	328
x264	hpel_filter	common/mc.c	184
x264	x264_me_search_ref	common/mc.c	173
x264	quant_4x4	common/quant.c	53

x264	sub4x4_dct	common/dct.c	112
x264	quant_trellis_cabac	common/rdo.c	419
blender	add_radiance	sss.c[1]	349
blender	ray_ao	rayshade.c[2]	2093
deepsjeng	feval	neval.cpp	1043
deepsjeng	see	see.cpp	19
deepsjeng	make	make.cpp	19
deepsjeng	RookAttacks	bitboard.cpp	545
deepsjeng	unmake	make.cpp	343
deepsjeng	remove_one_fast	search.cpp	368
deepsjeng	attacks_to	attacks.cpp	124
deepsjeng	gen	generate.cpp	159
deepsjeng	gen_captures	generate.cpp	398
imagick	MorphologyApply	magick/morphology.c	3827
imagick	MeanShiftImage	magick/feature.c	2108
imagick	SetPixelCacheNexusPixels	magick/cache.c	4732
imagick	GetOneCacheViewVirtualPixel	magick/cache-view.c	768
leela	FastBoard::kill_or_connect	FastBoard.cpp	1214
leela	FastBoard::update_board_fast	FastBoard.cpp	743
leela	FastBoard::self_atari	FastBoard.cpp	1271
leela	FastBoard::is_eye	FastBoard.cpp	805
leela	FastBoard::merge_strings	FastBoard.cpp	657
leela	FastBoard::save_critical_neighbours	FastBoard.cpp	1168
leela	FastBoard::add_pattern_moves	FastBoard.cpp	1908
leela	FastBoard::kill_neighbours	FastBoard.cpp	1105
leela	FastBoard::nbr_criticality	FastBoard.cpp	2044
leela	FastBoard::add_neighbour	FastBoard.cpp	202
nab	mme34	eff.c	3203
nab	heapsort_pairs	nblist.c	114
nab	nbond	eff.c	768
nab	searchkdtree	nblist.c	667
xz	bt_find_func	liblzma/lz/lz_encoder_mf.c	453
xz	bt_skip_func	liblzma/lz/lz_encoder_mf.c	521

[1]Full path: blender/source/blender/render/intern/source/sss.c
[2]Full path: blender/source/blender/render/intern/source/rayshade.c

Table A.4: Functions selected for Machine-10 E10-S10.

Benchmark	Function	Source File	Line
namd	pairlist_from_pairlist	ComputeNonbondedInl.h	32
namd	calc_pair_energy_fullelect	ComputeNonbondedUtil.h	352
povray	All_Sphere_Intersections	spheres.cpp	122
lbm	LBM_performStreamCollideTRT	lbm.c	262
x264	x264_pixel_satd_8x4	common/pixel.c	234
x264	get_ref	common/mc.c	232
imagick	MeanShiftImage	magick/feature.c	2108
imagick	MorphologyApply	magick/morphology.c	3827
imagick	SetPixelCacheNexusPixels	magick/cache.c	4732
nab	heapsort_pairs	nblist.c	114

Table A.5: Functions selected for Machine-10 E05-S05.

Benchmark	Function	Source 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	FastBoard::kill_or_connect	FastBoard.cpp	1214
leela	FastBoard::is_eye	FastBoard.cpp	805
leela	FastBoard::get_pattern_fast_augment	FastBoard.cpp	1356
nab	mme34	eff.c	3203
nab	heapsort_pairs	nblist.c	114
nab	searchkdtree	nblist.c	667

Table A.6: Functions selected for Machine-10 E02-S02.

Benchmark	Function	Source File	Line
perlbench	Perl_leave_scope	scope.c	759
perlbench	Perl_pp_multideref	pp_proto.h	161
perlbench	Perl_regexec_flags	regexec.c	2788
perlbench	Perl_pp_match	pp_proto.h	153
perlbench	Perl_pp_padsv	pp_proto.h	184
namd	pairlist_from_pairlist	ComputeNonbondedInl.h	32
namd	calc_pair_energy_fullelect	ComputeNonbondedUtil.h	352
namd	calc_pair_energy	ComputeNonbondedUtil.h	350
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_merge_fullelect	ComputeNonbondedUtil.h	363
namd	calc_self_energy_fullelect	ComputeNonbondedUtil.h	361
namd	calc_self_fullelect	ComputeNonbondedUtil.h	359
namd	calc_self_merge_fullelect	ComputeNonbondedUtil.h	362
namd	calc_self	ComputeNonbondedUtil.h	358
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
povray	Inside_Quadric	quadrics.cpp	283
povray	DNoise	texture.cpp	548
povray	All_Quadric_Intersections	quadrics.cpp	136
povray	Intersect_Light_Tree	lbuffer.cpp	1279
lbm	LBM_performStreamCollideTRT	lbm.c	262
omnetpp	cMessageHeap::shiftup	simulator/cmessageheap.cc	195
omnetpp	record[3]	indexedfileoutvectormgr.cc[4]	188
omnetpp	EtherMAC::handleMessage	model/EtherMAC.cc	202
omnetpp	cQueue::pop	simulator/cqueue.cc	304
omnetpp	cDatarateChannel::deliver	simulator/cdataratechannel.cc	129
omnetpp	cMessageHeap::insert	simulator/cmessageheap.cc	166
xalancbmk	NameDatatypeValidator::compare	NameDatatypeValidator.cpp	74
x264	x264_pixel_satd_8x4	common/pixel.c	234
x264	get_ref	common/mc.c	232

[3]Full name: cIndexedFileOutputVectorManager::record
[4]Full path: simulator/indexedfileoutvectormgr.cc

x264	mc_chroma	common/mc.c	263
x264	x264_pixel_sad_16x16	common/pixel.c	61
x264	pixel_hadamard_ac	common/pixel.c	328
x264	hpel_filter	common/mc.c	184
x264	sub4x4_dct	common/dct.c	112
x264	x264_me_search_ref	common/mc.c	173
x264	quant_trellis_cabac	common/rdo.c	419
x264	quant_4x4	common/quant.c	53
x264	x264_pixel_satd_4x4	common/pixel.c	209
blender	add_radiance	sss.c[5]	349
deepsjeng	feval	neval.cpp	1043
deepsjeng	see	see.cpp	19
deepsjeng	make	make.cpp	19
deepsjeng	attacks_to	attacks.cpp	124
deepsjeng	unmake	make.cpp	343
deepsjeng	gen	generate.cpp	159
deepsjeng	remove_one_fast	search.cpp	368
deepsjeng	eval	neval.cpp	1091
deepsjeng	FindFirstRemove	bits.cpp	49
deepsjeng	gen_captures	generate.cpp	398
imagick	MeanShiftImage	magick/feature.c	2108
imagick	MorphologyApply	magick/morphology.c	3827
imagick	SetPixelCacheNexusPixels	magick/cache.c	4732
leela	FastBoard::kill_or_connect	FastBoard.cpp	1214
leela	FastBoard::is_eye	FastBoard.cpp	805
leela	FastBoard::get_pattern_fast_augment	FastBoard.cpp	1356
leela	FastBoard::merge_strings	FastBoard.cpp	657
leela	FastBoard::add_pattern_moves	FastBoard.cpp	1908
leela	FastBoard::kill_neighbours	FastBoard.cpp	1105
nab	mme34	eff.c	3203
nab	heapsort_pairs	nblist.c	114
nab	nbond	eff.c	768
nab	searchkdtree	nblist.c	667
xz	lzma_mf_bt4_find	liblzma/lz/lz_encoder_mf.c	683
xz	lzma_decode	liblzma/lz/lz_decoder.c	284

[5]Full path: blender/source/blender/render/intern/source/sss.c