FITTCHOOSER: A Dynamic Feedback-Based Fittest Optimization Chooser

Arif Ali AP, Kévin Le Bon, Byron Hawkins and Erven Rohou
Univ Rennes, Inria, CNRS, IRISA
Email: {arif-ali.ana-pparakkal, kevin.le-bon, byron.hawkins, erven.rohou}@inria.fr

***Abstract*—Modern hardware features can boost the performance of an application, but software vendors are often limited to the lowest common denominator to maintain compatibility with the spectrum of processors used by their clients. Given more detailed information about the hardware features, a compiler can generate more efficient code, but even if the exact CPU model is known, manufacturer confidentiality policies leave substantial uncertainty about precise performance characteristics. In addition, the activity of other programs colocated in the same runtime environment can have a dramatic effect on application performance. For example, if a shared CPU cache is being heavily used by other programs, memory access latencies may be orders of magnitude longer than those recorded during an isolated profiling session, and instruction scheduling based on such profiles may lose its anticipated advantages. Program input can also drastically change the efficiency of statically compiled code, yet in many cases is subject to total uncertainty until the moment the input arrives during program execution.**

We have developed FITTCHOOSER to defer optimization of a program's most processor-intensive functions until execution time. FITTCHOOSER begins by profiling the application to determine the performance characteristics that are in effect for the present execution, then generates a set of candidate variations and dynamically links them in succession to empirically measure which of them performs best. The underlying binary instrumentation framework Padrone allows for selective transformation of the program without otherwise modifying its structure or interfering with the flow of execution, making it possible for FITTCHOOSER to minimize the overhead of its dynamic optimization process. Our experimental evaluation demonstrates up to 19% speedup on a selection of programs from the SPEC CPU 2006 and PolyBench suites while introducing less than 1% overhead. The FITTCHOOSER prototype achieves these gains with a minimal repertoire of optimization techniques taken from the static compiler itself, which not only testifies to the effectiveness of dynamic optimization, but also suggests that further gains can be achieved by expanding FITTCHOOSER'S repertoire of program transformations to include more diverse and more advanced techniques.

Keywords*—Dynamic Optimization; Code Generation; Compilers; Program Transformation; Binary Rewriting

I. INTRODUCTION

The introduction of advanced features like hardware counters and vector processing units in modern microprocessors can enable programs to run faster without requiring changes to the source code. For example, a vector processing unit can operate on an entire array of data in a single instruction. Programs can even be optimized at runtime by dynamically monitoring hardware counters [1], [2], [3], [4]. But software vendors must always take hardware compatibility into consideration before enabling these these advanced features in their applications. Many processor models do not support all the latest optimization features and will raise a hardware fault if a program attempts to invoke them. But even if the vendor could compile the program directly on each deployment machine separately—which is highly impractical—today's best commercial and open-source compilers often miss optimization opportunities. This is due in part to lack of public information about the low-level details of CPU features. Without a precise model of the processor's performance characteristics, the compiler resorts to heuristics for selecting such essential factors as the number of loop unrollings or the scheduling of load instructions [5]. Our experimental results in Section IV show that these heuristical models do not always make the best choice for a given program and hardware environment.

We have developed a dynamic optimization tool called FITTCHOOSER to overcome these limitations by generating variations of the program's machine code and empirically evaluating the variations to select the best performer. This iterative process allows FITTCHOOSER to find the most suitable optimization technique for a program's most processor-intensive functions in its current runtime environment. To account for potential changes in performance characteristics, which could for example be caused by expansion of a frequently traversed array beyond the capacity of the L3 cache, FITTCHOOSER continuously monitors its optimized functions and restarts the evaluation process when significant changes are observed.

Although our experiments show that FITTCHOOSER is efficient enough to recover its own overhead where it discovers effective optimizations for the target program, it may not always be practical to run the program under FITTCHOOSER. For example, SPMD programs run in parallel on all cores of a machine, while FITTCHOOSER anticipates that it can run on a separate core to mask the majority of its overhead. In such cases the user can conduct a preliminary tuning phase where FITTCHOOSER discovers the best optimizations for the machine, and then deploy those optimizations using our extension of the Linux loader called the FITTLAUNCHER. Although this approach does not benefit from the per-execution tuning of FITTCHOOSER, it does integrate the selected optimizations without the overhead of profiling and monitoring.

The current version of FITTCHOOSER explores a limited set of optimizations based on ordinary features of popular compilers, often making better use of those features than the compiler itself. Future enhancements to FITTCHOOSER could expand its repertoire to include advanced optimizations reported only in research, along with new experimental optimizations developed specifically for the tool. The remainder of this paper focuses on the FITTCHOOSER infrastructure and presents the currently available optimizations as a proof of concept that in our experience works in practice. Section II begins with an overview of FITTCHOOSER and Section III describes the implementation. We report the results of our experiments in Section IV, which include the overhead of FITTCHOOSER along with key examples of successful optimizations. Section V presents related work and Section VI concludes.

II. SURVIVAL OF THE FITTEST

The performance of a given execution of a program can be affected by a broad range of factors, making it difficult to determine in advance which optimization techniques may be the most advantageous. Many of these factors can be entirely unpredictable, for example if the program processes an input stream representing end-user activity, it may not be possible to predetermine the ideal optimizations for a given period of that input stream. Even if a compiler were to choose the ideal optimizations for a given execution scenario, the same compiled program could be executed in a slightly different scenario where other optimizations would improve performance. To bridge this gap between compile-time optimization and a concrete program execution, FITTCHOOSER employs dynamic instrumentation to generate and test various combinations of optimizations at the beginning of program execution and then transform the program to use the combination that empirically proves itself to be the most effective. This is implemented as a progression through three phases:

- *Initial Profiling*: Identify the 5 most processor-intensive functions within the current execution.
- *Optimization Pass*: Generate variations of those functions and dynamically link them into the running program, then iteratively profile each one for comparative effectiveness.
- *Cruise Control*: Dynamically link the variation that proved to be the most fit for the current execution.
  - Periodically monitor its performance and return to the Optimization Pass if significant changes are observed.

A. Initial Profiling

The key advantage of FITTCHOOSER over optimizing at compile-time is that it can precisely discover the program's performance characteristics, not just for a given machine, but also for a specific execution. This comes at the cost of runtime overhead to modify the machine code while the program is performing its tasks. To avoid squandering potential speedups, FITTCHOOSER profiles the application to identify the five functions in which the processor spends the majority of its time. These few *critical functions* are selected as exclusive candidates for optimization. Since this phase is never revisited, FITTCHOOSER must be configured with a long enough profiling period to accurately select the critical functions.

B. Optimization Pass

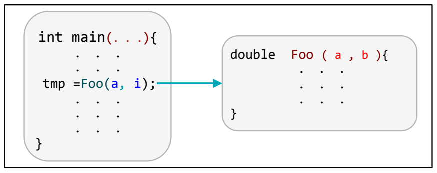
This phase begins with an analysis of each critical function to determine which program transformations can potentially be advantageous. For example, functions containing loops are typically candidates for loop unrolling and loop tiling. Conversely, functions containing `static` variables are not eligible for these optimizations because of the difficulty in preserving the value of the variable across different variations of the function. The optimization repertoire of FITTCHOOSER is presented in detail in Section III.

For each candidate optimization, a new version of the selected function is generated and injected into the running process. To compare the performance of the variations, we inject a meta-function `monitor` that acts both as a dispatcher and a timer. The monitor rotates between the injected variations in round robin fashion to maintain timing fairness. Each variation is allocated a fixed (configurable) number of invocations per round, and evaluation continues until the total number of invocations reaches a fixed (configurable) threshold. At the end of this evaluation period, the monitor functions are retired by patching calls directly to the best-performing variation.

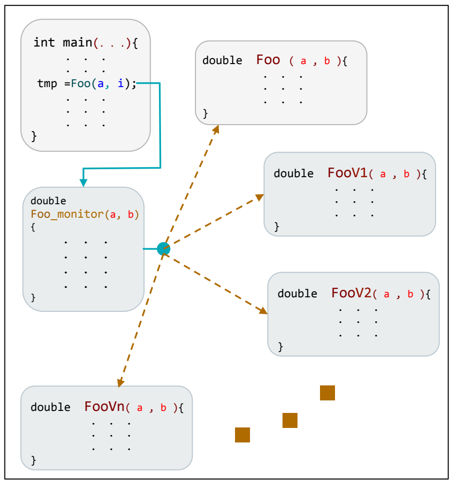
The pseudocode in Listing 1 illustrates a common scenario where a single progression through the variations would result in unfair evaluation. Since the number of iterations of the `for` loop in function `Foo` depends on parameter `b`, the value of the parameter `b` affects the running time of the function `Foo`. The first 100 calls to the function have an average of 50 iterations, while the second 100 calls have an average of 150 iterations. Suppose we run the first version for the first 100 calls and the second version for next 100, then comparing the average running time of them to find the fastest is unfair. Instead, executing them in a round robin fashion with a quanta of 10 calls makes it more comparable.

While there are other ways to maintain fairness of evaluation, the monitor function has been implemented with a round robin strategy to avoid complications with low-level timing measurement. An alternative approach could measure flops, but this generally requires hardware counters that may not be available on older processor models. Another option would be to measure instructions per second, but this will be inaccurate for optimizations that reduce the number of executed instructions (for example, loop unrolling may eliminate a significant number of branch instructions).

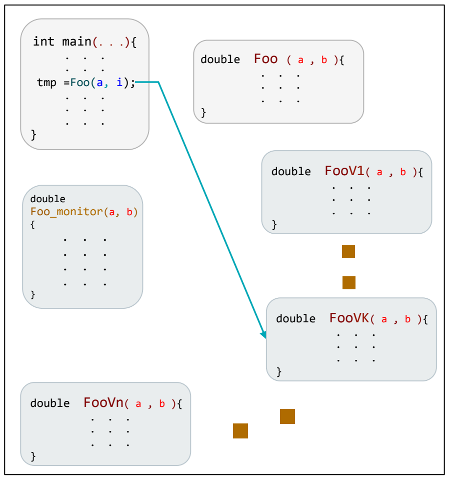
Figure 1 illustrates the Optimization Pass by depicting the execution of function `Foo` when the example program given in Listing 1 is executed under FITTCHOOSER. The original (statically compiled) control flow is shown in Figure 1a, and the dynamically linked `Foo_monitor` function appears in Figure 1b. After choosing the best variation, FITTCHOOSER bypasses the monitor by linking call sites directly to it, as shown in Figure 1c, and then goes on Cruise Control.



(a) Original call direct to `Foo()`.



(b) Call redirected to the monitor function, which dispatches to the injected variations.



(c) Call direct to the fittest variation.

Figure 1: Progression of the Optimization Pass.

Listing 1 Example Function

```
 double Foo(unsigned int a[], unsigned int b)
    for i = 1 to  b
        c += a[i]/b;
  return c;

  int main (int argc, char *argv[] )
    _ _ _
   for i = 1 to  10000
      _ _ _
      temp = Foo(a,i);
      _ _ _
   _ _ _
```

C. Cruise Control

Application behavior may change during execution such that our selected program transformations may no longer be optimal. To maintain performance through these changes, FITTCHOOSER remains on Cruise Control throughout the execution of the program, periodically evaluating the optimized functions. If significant changes are observed, FITTCHOOSER revisits the Optimization Pass in search of the best variations for the present conditions.

III. FITTCHOOSER

The three-phase strategy for finding and linking the fittest optimizations is coordinated by the FITTCHOOSER application, which runs in its own separate process. Inter-process communication is facilitated by the binary instrumentation framework Padrone [6], which supports selective instrumentation of a target process while minimizing interference with its native flow of execution. Section III-A presents Padrone in more detail and makes a case for its fitness as the foundation of FITTCHOOSER. Section III-B moves on to the implementation of FITTCHOOSER, and Section III-C describes the lightweight deployment alternative FITTLAUNCHER.

A. Padrone

Since our goal is to improve performance while monitoring and modifying the program, it is essential for FITTCHOOSER to minimize its own overhead. While there are many tools that can facilitate the instrumentation, to our knowledge Padrone is the least intrusive among them and therefore the most advantageous for conserving speedups. Where a typical binary translator takes full control of a program and translates every executed instruction, Padrone provides comparable instrumentation on a selective basis, modifying only the instrumented bytes. An alternative approach would be to compile the instrumentation directly into the target application, but this can change critical performance factors such as code layout, and introduces the challenge of integrating with the build system of every target application. A self-contained tool like Padrone is more practical, communicating with the target binary over the the Linux `ptrace` API like an interactive debugger.

Padrone instruments the target process by injecting code changes, which includes modifying existing program instructions and/or generating code to a dynamically allocated code cache. It is also possible to inject a shared library via `dl_open()`. To link a new function into the running program, Padrone inserts a `trap` at the start of the original function to identify incoming calls (by checking the return address at the trap). After modifying the operand of the incoming call instruction, subsequent invocations of that call site will go directly to the new target function.

Padrone offers substantial advantages over conventional binary translation, especially in the context of performance analysis and dynamic optimization:

- Binary translators have a baseline overhead of at least 12% on the SPEC CPU 2006 benchmark suite [7], increasing to 30% or more for desktop applications [8], and inflating up to 17× where a JIT engine is involved [9]. In contrast, the baseline overhead of Padrone is negligible, consisting of just one interruption for `ptrace` attach.
- Padrone does not require global monitoring of system calls or standard library calls to patch up their effects because its selective instrumentation makes it inherently transparent to the target program.
- Binary translators typically replace the `ret` instruction with a `push/jmp` that interferes with hardware optimizations for call/return symmetry.
- Padrone's selective instrumentation allows greater control over the alignment and colocation of injected code by reducing pressure to consolidate the code cache.
- Many important hardware performance counters are local to a CPU core, allowing for accurate measurements even while other cores are highly active. This advantage is lost under in-process binary translation where the activity of the translator pollutes the local counters. Padrone is able to monitor performance counters in the target process via the PAPI function `PAPI_attach()` while limiting its own footprint on the monitored core to the relatively lightweight `ptrace` calls.

While it is possible to compile FITTCHOOSER directly into the target application, this is highly impractical for most deployed software, and impossible for legacy binaries that were compiled before Padrone was available.

B. FITTCHOOSER

We implement FITTCHOOSER in plain `C` using Padrone's API for introspection and instrumentation of the running process. To optimize a program with FITTCHOOSER, the user first launches the program and then passes the process ID to FITTCHOOSER, which connects via `ptrace` and begins the Initial Profiling phase. In its current stage of development, our FITTCHOOSER prototype also requires the user to provide the LLVM IR [10] of the target program. This inconvenience can be replaced by a technique to lift the program's machine code to LLVM IR. Previous work by Hallou et al. [11] showed that decoding with the McSema infrastructure [12] yields LLVM IR suitable for optimizations as complex as vectorization. A similar approach could leverage the decoder of the binary translator HQEMU [13], which presents as its fundamental contribution the transformation of the internal QEMU IR to LLVM IR such that the LLVM compiler can be used to optimize translated code on the fly. Given this future enhancement, FITTCHOOSER would no longer require the user to provide any information about the target application.

1) Candidate Optimizations: The efficiency of FITTCHOOSER and its underlying framework Padrone are essential for realizing performance gains from an optimization that is constructed entirely at runtime. But the pivotal component of FITTCHOOSER is its code generator that produces the optimization candidates. Given an extensive repertoire of powerful optimizations, the potential speedup depends mainly on the selection of candidates that have a high probability of (a) significantly increasing performance of the target function while (b) maintaining near-native performance even in an unexpected worst-case scenario. While our current experimental results operate on a limited optimization vocabulary that focuses on standard loop unrolling, the integration of the LLVM compiler provides FITTCHOOSER with easy access to a broad range of optimizations available from the LLVM community. As more advanced techniques are incorporated into FITTCHOOSER, its analysis of the target function will need to be increasingly effective in identifying the most promising avenues of optimization while recognizing potential pitfalls that could incur unacceptable overheads during evaluation.

The analysis of the target function can potentially be complemented by dynamic profiling of performance counters (where available). For example, if a group of optimizations aims to reduce the frequency of a certain hardware operation, a dynamic profile of corresponding hardware counters could enable FITTCHOOSER to accurately estimate the potential of those optimizations for the current execution. Research has explored the use of hardware counters in profile-guided optimization [14], including dynamic compilers such as JIT engines [15], but these efforts report significant difficulty in correlating hardware events with specific program code fragments. These problems do not occur for FITTCHOOSER because, instead of speculating about the significance of hardware event counts, it can explore a hypothesis about potential optimizations by simply generating an exploratory variation of the target function and empirically observing the change (or lack thereof) in hardware events.

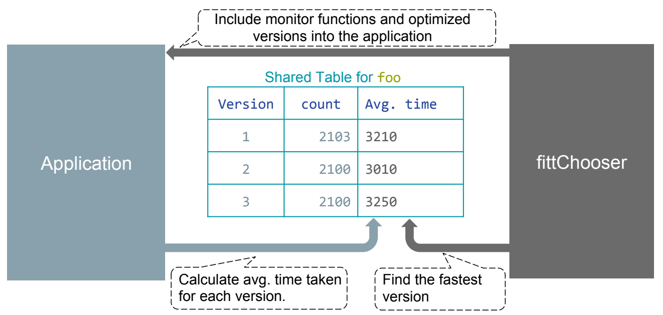
2) Monitor Functions: While the round robin dispatch of the monitor functions is relatively straightforward, two interesting challenges arise where the functions are integrated into the target program. The first is to compare the performance of the injected variations without encumbering the target program. The monitor function could easily perform the comparison directly, but this can involve a significant amount of computation, especially when there is analysis involved in determining what action to take next. So instead, the monitor function records the average execution time of each round to a table

Variation	Count	Timings
1	2103	3210
2	2100	3021
3	2100	3310
Original	2100	3250

TABLE I: Example of the shared Monitoring table.

that is shared between the target program and FittChooser. An example of its contents is depicted in Table I. The table is hosted in a System V shared memory segment, and shared between the two processes as shown in Figure 2. In our current implementation, synchronization is not required for table access because FittChooser waits until the end of the Optimization Pass and reads the entire table after the monitor functions have been removed. In the case that future enhancements involve intermediate evaluation of the variations, for example to adjust them during the Optimization Pass, it may become necessary to introduce a locking scheme.

The second challenge involves the pass-through of the return value from the target function to its caller within the target program. It would be simple if the monitor could be inserted prior to the call and simply change its target, but this is not possible because the monitor must stop its internal timer when the function returns, and at the end of a round it must also update the shared table with the observed average execution time. Instead, the monitor function stores the return value from the target function, performs the necessary computations and updates, and then returns the value. This is difficult to do in a generic manner because the return value may take the form of a struct which requires a copy through memory. Since the monitor functions are generated from an LLVM IR template that requires an accurate declaration of the return type, our prototype is limited to target functions that return a scalar data type (including pointer types). This limitation could more easily be alleviated by an assembly implementation of the monitor functions, or by a wrapper function for the monitor that is written in assembly.


Include monitor functions and optimized versions into the application
Application
fittChooser
Shared Table for foo
Calculate avg. time taken for each version.
Find the fastest version

Version	count	Avg. time
1	2103	3210
2	2100	3010
3	2100	3250

Figure 2: Communication through the shared Monitoring Table.

C. FittLauncher

The user may prefer to conduct a preliminary discovery phase and later incorporate the resulting optimizations without running FittChooser. For this scenario we provide the FittLauncher, which is an extension of the Linux loader that links a set of pre-defined function variations into a program at load time. Since this deployment model operates in-process and only takes action at module load time, it eliminates the separate FittChooser process along with its overhead. There are also usability advantages:

- The FittLauncher is compatible with GDB, whereas FittChooser introduces a conflict over the `ptrace` API which only supports one connection at a time.
- FittChooser requires complex configuration of thresholds and other special knowledge, whereas the FittLauncher is a simple command-line prefix.

This two-phase approach with FittLauncher also opens the door to a more aggressive configuration of optimizations in FittChooser. For important programs that warrant a dedicated optimization effort, the user may conduct an exploratory phase in which FittChooser is configured to take greater risks. The performance of these exploratory runs may be very poor over all—since many of the attempted variations will be unsuccessful—but FittChooser will be able to evaluate a broader range of candidates that may lead to discovery of unexpectedly effective variations. After configuring the FittLauncher to incorporate the best performers into the program at load time, the program can benefit from the speedup without any further overhead from FittChooser.

The FittLauncher can either be installed in the operating system to take effect for all programs, or it can be invoked selectively by passing the name of the program to launch along with its arguments (the default Linux loader supports the same usage model). As FittLauncher loads program modules into memory, it consults a database of installed optimizations. If any are found, it invokes `mmap` to request a region of memory near the corresponding module and populates it with the optimized function variations. Then FittLauncher links each function by inserting a 5-byte hook in the prologue of the original. To eliminate overhead from the hook, a more advanced implementation could identify the function callers and simply change their target operand, as in FittChooser. But our experimental results show that even with the hook, FittLauncher imposes less than 0.2% overhead (geometric mean) across the SPEC 2017 benchmark suite [16].

To maintain compatibility with the host Linux platform, we provide a Python script to generate the FittLauncher from the existing system default loader. The script adds an executable section to the end of the loader and installs a callback hook in the main executable section where internal accounting is performed for a newly loaded module. We implement the FittLauncher functionality as a static library in plain `C` and splice it into the appended executable section of the loader. To avoid dependencies on loaded modules (which are generally not available to the loader itself!) the FittLauncher generator script identifies useful symbols such as `open` (for opening files) and `strcpy` in the original loader and statically links them to the FITTLAUNCHER internal functions as necessary.

IV. RESULTS

We conducted our performance experiments on a 2.7GHz Intel Core i7 Broadwell desktop supporting SIMD and AVX2 with an L3 cache of 4MB and 16GB RAM. The machine runs Linux 3.19 and our benchmarks are compiled with LLVM version 3.7.0 at level `-O3`. We use `taskset` to pin the application to a single core and the `time` command for measurement.

Our experiments focus on a subset of benchmarks from PolyBench [17] and *SPEC CPU 2006* [18] benchmark suites. The subset is partly necessary because Padrone only supports programs written in `C`. More importantly, our main goals in this evaluation are to (a) show the potential speedups that FITTCHOOSER can discover, and (b) to demonstrate that FITTCHOOSER can apply these optimizations efficiently, without squandering the speedup. For many benchmarks in these two suites, there is either no function compatible with FITTCHOOSER (for example because of a complex return type), or the benchmark contains no candidate functions for our limited repertoire of program transformations. So we focus our experiments on benchmarks having candidate critical functions under the proposition that future versions of our tool will be able to successfully optimize the excluded benchmarks with the benefit of an expanded repertoire.

We make two minor adjustments to The PolyBench suite because it calls the critical function only once, whereas FITTCHOOSER is designed to optimize iterative programs. Our changes include (1) a `for` loop to call the function one million times and (2) `__attribute__((noinline))` to prevent inlining of the critical function.

A. Overhead

Although there is significant computational overhead for both the Initial Profiling and Optimizaton Pass, the majority of the overhead is masked by performing the processor-intensive tasks in the parallel FITTCHOOSER process. Upon reaching Cruise Control the periodic profiling has negligible overhead because it is invoked sparsely and for a short duration. Figure 3 shows the overhead of (a) profiling alone and (b) profiling and monitoring combined across four benchmarks from the PolyBench suite. The configuration for profiling alone focuses on the top critical function and includes 3 sessions of 5, 10 and 20 seconds with a frequency of 100Hz, 200Hz and 400Hz, respectively. Monitoring is configured to terminate at a threshold of 20,000 total invocations of the critical function, and deploys a *null optimization* which simply contains a copy of the original critical function (compilation time of the copy is included in these results). This represents a median scenario where the attempted variations collectively perform roughly the same as the original—performance can deteriorate if more aggressive (and less reliable) optimizations are attempted. As

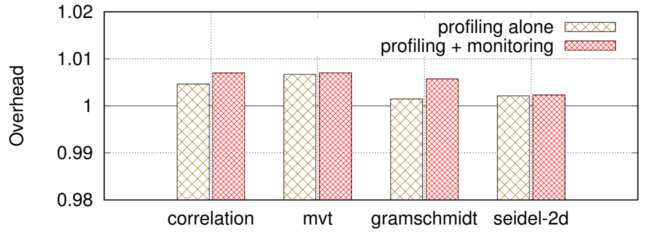


Figure 3: Overhead of FITTCHOOSER.

shown in the figure, the overhead is less than 1% throughout the course of the benchmark in all 4 cases.

B. Speedup

Figure 4 reports the overall speedup obtained for the selected subset of the PolyBench and SPEC CPU 2006 benchmark suites. This includes the Initial Profiling and the full Optimization Pass with all associated overheads. Both the benchmarks and the injected variations are compiled at LLVM optimization level `-O3`. The configuration attempts 13 variations of the top critical function in each application. The first variation is produced by recompiling the IR with only the `march=native` flag. The remaining 12 variations progressively assign the `-loop-unroll` flag from 2 to 24 (stepping by 2). In the Optimization Pass, each variation is invoked at least 100 times with a quanta of 10, and timing is measured over the last 100 invocations after discarding the highest 10 results to compensate for noise. Figure 5 depicts the performance of these optimization flags on four PolyBench programs.

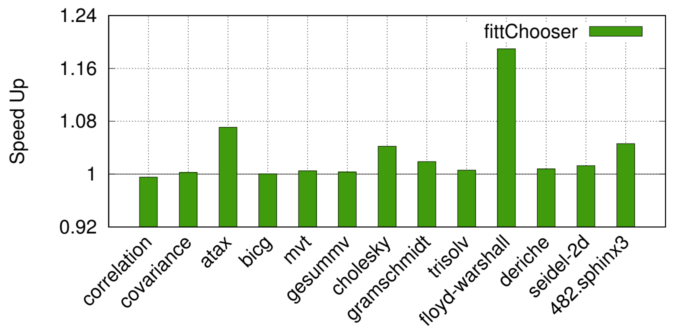


Figure 4: Overall speedup under FITTCHOOSER.

We experienced a slight slowdown for the `correlation` benchmark. It showed an overhead of 0.6 % in Figure 3 and slowdown of 0.4 % in Figure 4, indicating that the speedup created by the optimized versions did not recover the overhead of the trials. Table II shows the standard deviation of the results. While the majority of the benchmarks cannot be improved beyond the `-O3` optimization level, several benefit greatly from FITTCHOOSER: `floyd_warshall` is 19 % faster, `atax` 7 % and `cholesky` 4 %.

Figure 5 shows that FITTCHOOSER selects different optimizations for different programs. For example,

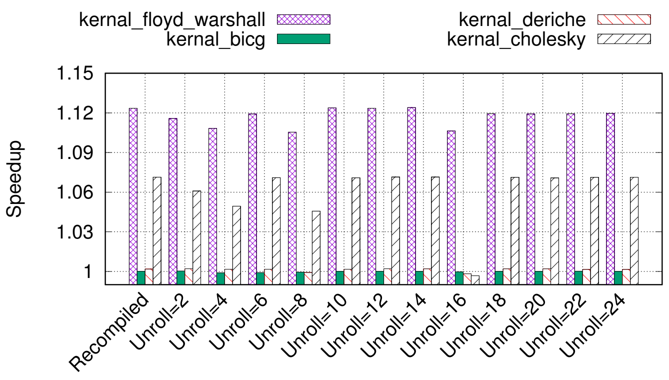


Figure 5: Performance of critical function variations.

Benchmark	Standard Deviation
correlation	0.49%
covariance	0.25%
atax	0.15%
bicg	0.01%
mvt	0.13%
gesummv	0.02%
cholesky	0.05%
gramschmidt	0.23%
trisolv	0.01%
floyd-warshall	0.07%
deriche	1.20%
seidel-2d	0.01%

TABLE II: Standard Deviation

`kernel_floyd_warshall` performs best with a loop unrolling of 14 whereas `kernel_cholesky` prefers 12. In cases where the performance is almost equal for all unrolling factors, FITTCHOOSER may assign a different variation depending on the exact performance observed during the execution. For example, among 10 executions of `kernel_floyd_warshall`, FITTCHOOSER assigns the 8^{th} version six times, the 1^{st} version three times and the 7^{th} version once.

We also observe that some applications are drastically improved simply by recompilation on the target machine using the default `-O3` optimizations. This still indicates an advantage of FITTCHOOSER over the conventional software distribution model where compilation is performed once at the vendor's site (similarly to [19]). No matter how trivial or sophisticated the source of the speedup, the dynamic optimization model finds increasing importance in today's rapidly expanding landscape of computing resources. Applications designed for desktop computers are commonly run on cloud servers, virtualization platforms and even mobile devices, introducing performance characteristics that can vary dramatically from the machine where the code was compiled. To the best of our knowledge, the only way to reliably tune application performance in such an environment is to optimize at the point of execution, which is where FITTCHOOSER excels.

C. FITTLAUNCHER

Since the expected usage of FITTLAUNCHER is to install optimizations for the program's most critical functions, we prepare our evaluation of FITTLAUNCHER by installing a *null optimization* for the top critical function of each program in the SPEC 2017 benchmark suite (as reported by Linux `perf`). We observe a geometric mean of 0.124% overhead, which falls below the standard deviation of 0.178% across the native executions of the suite.

V. RELATED WORK

Dynamic Binary Rewriting Pin [20] is a dynamic binary instrumentation framework with a flexible API that has enabled development of a rich set of Pintools for architecture exploration, emulation and security. Because Pin focuses on instrumentation and analysis, it always runs the target program from a copy in its code cache. DynamoRIO [21] is a similar tool that focuses on efficiency and provides a simple lightweight API to clients. It can execute the target program entirely from its code cache, or partly native, and can consolidate cached code into traces for efficiency. Valgrind [22] focuses more on its instrumentation capabilities than performance, and the framework is designed for *heavyweight* tools: every instruction is instrumented, and a high volume of information about the target program's execution is collected. The novelty of Valgrind is the use of *shadow values* [23] for register and memory locations, yielding a more powerful analysis at the cost of higher overhead.

Iterative Compilation is similar to FITTCHOOSER in that it addresses the performance issues that arise from detailed hardware characteristics and transitory factors of the runtime environment. The key idea is to identify local minima by producing many versions of the same program and running them on various platforms to identify the best overall performers. Iterative compilation has been advanced by machine learning techniques that are broadly covered in a survey by Ashouri et al. [24]. Our work takes the same basic approach, but we apply it at runtime. By doing so, we concentrate only on the most time consuming functions, and we can easily adjust our optimizations for the performance characteristics that are directly affecting the current execution of the program.

JIT Compilers apply different levels of optimizations to functions when they become time consuming (see for example the discussion of Oracle's HotSpot compiler [25]). Their purpose is different from ours: they want to spend time optimizing functions only when the chances are high to recoup the time in future execution time. Optimizations available in each level are fixed, while we explore many variants.

JIT technology with C/C++ [19] discusses about dynamic optimization by using both native executable file and intermediate representation (IR) file of the program. During execution, they recompile hot methods from IR file using a Java JIT compiler and the recompiled versions are stored in a code cache. With the help of a trampoline created at the beginning of the original function body, the function calls are redirected to the new recompiled version. This work applies a similar recompilation technique to FITTCHOOSER, but creates only one version of the function and does not evaluate its performance against the statically compiled version in the original binary.

Dynamic function specialization [26] is limited to specialization based on argument values, whereas FITTCHOOSER follows in the vein of iterative compilation and can apply any optimization, provided that its analysis of the LLVM IR is sufficient for the corresponding program transformations.

VI. CONCLUSION

Detailed information about hardware performance characteristics can improve compiler optimizations, but applications are typically compiled for use on many different architectures having a broad range of performance characteristics. Transitory factors of the runtime environment can also affect application performance. We propose FITTCHOOSER to dynamically evaluate the fitness of candidate optimizations for a program's critical functions and then replace the original functions on the fly, all without restarting the program. Experimental evaluation of FITTCHOOSER on important industry benchmarks demonstrates up to 19% speedup even with a limited repertoire of program transformations, suggesting even more gains may be possible as more sophisticated optimization techniques are incorporated into FITTCHOOSER.

REFERENCES

[1] J. Lehr, "Counting performance: hardware performance counter and compiler instrumentation," in *Informatik 2016, 46. Jahrestagung der Gesellschaft für Informatik, 26.-30. September 2016, Klagenfurt, Österreich* (H. C. Mayr and M. Pinzger, eds.), vol. P-259 of *LNI*, pp. 2187–2198, GI, 2016.

[2] B. Wicht, R. A. Vitillo, D. Chen, and D. Levinthal, "Hardware counted profile-guided optimization," *CoRR*, vol. abs/1411.6361, 2014.

[3] F. Schneider and T. R. Gross, *Using Platform-Specific Performance Counters for Dynamic Compilation*, pp. 334–346. Springer Berlin Heidelberg, 2006.

[4] N. Watkinson, A. Shivam, Z. Chen, A. Veidenbaum, and A. Nicolau, "Using hardware counters to predict vectorization," 2017.

[5] R. S. Machado, R. B. Almeida, A. D. Jardim, A. M. Pernas, A. C. Yamin, and G. G. H. Cavalheiro, "Comparing performance of C compilers optimizations on different multicore architectures," in *2017 International Symposium on Computer Architecture and High Performance Computing Workshops (SBAC-PADW)*, pp. 25–30, Oct 2017.

[6] E. Riou, E. Rohou, P. Clauss, N. Hallou, and A. Ketterlin, "PADRONE: a Platform for Online Profiling, Analysis, and Optimization," in *DCE 2014 - International workshop on Dynamic Compilation Everywhere*, (Vienna, Austria), Jan. 2014.

[7] B. Hawkins, B. Demsky, D. Bruening, and Q. Zhao, "Optimizing binary translation of dynamically generated code," in *Proceedings of the 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '15, (Washington, DC, USA), pp. 68–78, IEEE Computer Society, 2015.

[8] S. Sinnadurai, Q. Zhao, and W.-F. Wong, "Transparent runtime shadow stack: Protection against malicious return address modifications." http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.120.5702&rep=rep1&type=pdf, 2006.

[9] B. Hawkins, B. Demsky, and M. B. Taylor, "Blackbox: Lightweight security monitoring for cots binaries," in *Proceedings of the 2016 International Symposium on Code Generation and Optimization*, CGO '16, (New York, NY, USA), pp. 261–272, ACM, 2016.

[10] C. Lattner and V. Adve, "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation," in *International Symposium on Code Generation and Optimization (CGO'04)*, 2004.

[11] N. Hallou, E. Rohou, and P. Clauss, "Runtime vectorization transformations of binary code," *International Journal of Parallel Programming*, pp. 1–30, 2016.

[12] A. Dinaburg and A. Ruef, "McSema: Static translation of x86 instructions to LLVM." https://blog.trailofbits.com/2014/06/23/a-preview-of-mcsema/. Accessed: 2016-11-02.

[13] D.-Y. Hong, C.-C. Hsu, P.-C. Yew, J.-J. Wu, W.-C. Hsu, P. Liu, C.-M. Wang, and Y.-C. Chung, "Hqemu: A multi-threaded and retargetable dynamic binary translator on multicores," in *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, CGO '12, (New York, NY, USA), pp. 104–113, ACM, 2012.

[14] D. Chen, N. Vachharajani, R. Hundt, S.-w. Liao, V. Ramasamy, P. Yuan, W. Chen, and W. Zheng, "Taming hardware event samples for fdo compilation," in *Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '10, (New York, NY, USA), pp. 42–52, ACM, 2010.

[15] F. Schneider and T. R. Gross, "Using platform-specific performance counters for dynamic compilation," in *Languages and Compilers for Parallel Computing* (E. Ayguadé, G. Baumgartner, J. Ramanujam, and P. Sadayappan, eds.), (Berlin, Heidelberg), pp. 334–346, Springer Berlin Heidelberg, 2006.

[16] spec.org, "SPEC CPU2017." https://www.spec.org/cpu2017.

[17] L.-N. Pouchet, "PolyBench/C v4.1: the polyhedral benchmark suite." http://web.cse.ohio-state.edu/~pouchet.2/software/polybench/.

[18] J. L. Henning, "SPEC CPU2006 benchmark descriptions," *SIGARCH Comput. Archit. News*, vol. 34, pp. 1–17, Sept. 2006.

[19] 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 Trans. Archit. Code Optim.*, vol. 10, pp. 59:1–59:25, Dec. 2013.

[20] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: building customized program analysis tools with dynamic instrumentation," 2005.

[21] D. Bruening, *Efficient, Transparent, and Comprehensive Runtime Code Manipulation*. PhD thesis, MIT, Sept. 2004.

[22] N. Nethercote and J. Seward, "Valgrind: a framework for heavyweight dynamic binary instrumentation," in *PLDI*, pp. 89–100, 2007.

[23] N. Nethercote, "Dynamic binary analysis and instrumentation," Tech. Rep. UCAM-CL-TR-606, University of Cambridge, Computer Laboratory, Nov. 2004.

[24] A. H. Ashouri, W. Killian, J. Cavazos, G. Palermo, and C. Silvano, "A survey on compiler autotuning using machine learning," *arXiv preprint arXiv:1801.04405*, 2018.

[25] M. Paleczny, C. Vick, and C. Click, "The Java HotSpot^{TM} Server Compiler," in *Proc. of the Java Virtual Machine Research and Technology Symposium*, (Monterey, CA, USA), Apr. 2001.

[26] A. Ap and E. Rohou, "Dynamic function specialization," in *International Conference on Embedded Computer Systems: Architectures, MOdeling and Simulation*, 2017.