International Journal on Science and Technology (IJSAT)
E-ISSN: 2229-7677 ● Website: www.ijsat.org ● Email: editor@ijsat.org

Real-time Compilation and Performance Monitoring for High-Performance Systems

Aadithya P Goutham[1], Jeeva N.S[2], Dr. Godfrey Winster S[3]

[1,2]Student, Department of Computing Technologies, SRM Institute of Science and Technology, Kattankulathur, India
[3]Associate Professor, Department of Computing Technologies, SRM Institute of Science and Technology, Kattankulathur, India

Abstract

High Performance Computing applications are diverse and they operate in dynamic environments. This requires a shift in compilation techniques from static, hardcoded algorithm-driven approaches to dynamic, real-time optimizing strategies. Current compiler methodologies primarily rely on static code analysis to apply optimizations during the compilation phase. Optimization techniques such as loop unrolling, vectorization, and function inlining are only effective for predictable workloads. However, they lack the adaptive ability in changing runtime conditions and hardware variability. System specific optimizations can be manually applied by the programmer, but this makes the code less portable and requires rewrite of entire programs, thus increasing the cost of maintenance. Our approach includes a real-time performance monitoring system that can trigger a recompilation dynamically to change code execution patterns. Runtime feedback is used to identify bottlenecks such as core overutilization, cache inefficiencies, and memory bottlenecks. A distinctive feature of the system is feedback-driven real-time compiler optimization. Whereas the performance benefits of dynamically compiled code is offset by the overhead incurred from the monitoring and recompilation system, the overall efficiency of the program throughout its runtime improves incrementally over each iteration of dynamically recompiled code. This efficiency improvement can also lead to energy savings in terms of reduction in wasted computational resources. The work presented here lays the foundations for adaptable and feedback-driven compiler optimization strategies.

Keywords: High Performance Computing (HPC), Just-in-time (JIT) Compiler, Low Level Virtual Machine (LLVM), Intermediate Representation (IR), GNU Compiler Collection (GCC), Recompilation and Optimization Decision Algorithm (RODA)

1. Introduction

High-Performance Computing (HPC) systems are fundamental to a wide array of resource-demanding applications, such as scientific simulations, artificial intelligence and big data analytics. These normally function in dynamic and heterogeneous environments, where conventional static compilation techniques may leave room for performance improvements. While static optimization techniques such as loop unrolling, vectorization, and function inlining are applicable for predictable workloads, they lack adaptability to real-time conditions with changing runtime conditions and variations in hardware. This

situation necessitates a major shift in compiler design towards dynamic, feedback-driven and efficient optimization methodologies.

Dynamic compilation has surfaced as a viable solution over time to bridge the disparity between static analysis and runtime variability. Utilizing runtime-invariant data allows dynamic compilers to implement sophisticated optimizations such as just-in-time specialization, branch elimination, and speculative execution, thereby significantly improving code execution efficiency. However, the overhead associated with runtime instrumentation (collection of metrics) and triggering of a recompilation will likely negate these performance enhancements and present a challenge to efficiency. Thus far, research in this area has investigated strategies to handle the overhead through development of lightweight runtime compilers and pre-optimized machine-code templates, with some of these indicating performance improvements ranging from 1.2x to 1.8x.

Integration of runtime performance monitoring into a dynamic compilation system adds a new layer of adaptability. Feedback-directed optimization frameworks in existence, such as PEAK, demonstrate the possibility of iterative tuning to improve program performance across variable optimization scenarios. These systems use decision algorithms to determine critical code segments and perform optimizations effectively, resulting in noticeable improvements in execution speed and reduction in tuning time. Similarly, frameworks like ApproxTuner illustrate the potential of predictive modeling in the same context of dynamic compilation.

Our system introduces an architectural overview that integrates real-time performance monitoring with dynamic recompilation. Unlike traditional methodologies, we seek to use an iterative process that incrementally enhances program execution patterns over time. This enhances runtime efficiency and minimizes computational waste in the form of power consumption, hence promoting energy conservation—an important aspect in high performance computing (HPC) environments where power efficiency dictates operational costs. Key to this framework is a feedback-driven optimization loop, which monitors the runtime environment constantly and triggers a recompilation of code whenever potential for optimization is detected.

The issues with runtime variability and hardware diversity are addressed by our proposed system and it demonstrates feasibility for future adaptable compiler technologies. This paper details the system architecture, underlying methodology and experimental validations, laying the foundation for future developments in real-time compilation techniques tailored for dynamic computing contexts.

2. Literature Review

The research work ApproxTuner: A Compiler and Runtime System for Adaptive Approximations aims to present accuracy-aware optimization of tensor-based applications while requiring only high-level end-to-end quality specifications. The key contribution in ApproxTuner is a novel three-phase approach to approximation-tuning that consists of development-time, install-time, and run-time phases. It addresses the challenge of automatically selecting, configuring and tuning the parameters for combinations of approximation techniques while meeting end-to-end requirements on energy, performance and accuracy. [1]

The research work Fast, Effective Dynamic Compilation addresses the effectiveness of dynamic compilation while keeping in view the challenges in implementing it. The work also presents findings on the effective utilization of static compilation methods to optimize simple code segments while using dynamic compilation to address programmer annotated code segments with code templates. The research targets general purpose imperative programming languages, like C. Their dynamic compilation system is composed of both a static and a dynamic compiler. To achieve fast dynamic compile times, the static compiler produces pre-compiled machine-code templates, whose instructions contain holes that will be filled in with run-time constant values. [2]

The research work Compiler Technology for Parallel Scientific Computation presents an approach based on a program decomposition, parallel code synthesis, and run-time support for parallel scientific computation. The program decomposition is guided by the source program annotations provided by the user. The synthesis of parallel code is based on configurations that describe the overall computation as a set of interacting components. The compiler-generated code provides runtime support through redistribution of computation and data during object program execution. Techniques such as data alignment, operator placement, wavefront determination and memory optimization are applied to parallel code. [3]

The research work Exploiting Parallelism for Energy Efficient Source Code High Performance Computing experimentally shows that energy efficiency is reduced by many factors, such as optimal architecture utilization, poor compilation optimization, to name a few. It presents a methodology that exploits parallelism, inherent in multimedia DSP applications, as well as in multimedia DSP processors and includes profile based compilation-approach which makes the source-to-source transformation more energy efficient. [4]

The research work Power-Aware Compilation Techniques For High Performance Processors presents findings on power consumption from the perspectives of register spilling, functional unit usage in software-pipelined loops and memory accesses due to cache misses. These form the groundwork for energy efficient compilation techniques and the metrics that are of most interest during compilation. [5]

The research work Compiler Transformations for High-Performance Computing aims to present a comprehensive overview of the important high-level program restructuring techniques for imperative languages such as C and Fortran. Transformations for both sequential and various types of parallel architectures are covered in depth. Major transformations include data-flow based loop optimization, loop reordering, loop restructuring, loop replacement, memory access, partial evaluation, redundancy elimination and procedure call transformation. Various transformation frameworks are also illustrated in detail with appropriate benchmark results. [6]

The research work A Script-Based Autotuning Compiler System to Generate High-Performance CUDA Code presents a novel compiler framework for CUDA code generation. The compiler structure is designed to support autotuning and a transformation strategy generator, an optimizer that yields performance

transformation recipes. These comprise a search space of possible implementations. This system has been demonstrably better than manually tuned libraries and GPU compilers. [7]

3. Architecture of Real-Time Performance Monitor

A. HPC Cluster Node

The system targets High-Performance Computing environments with the need for dynamic optimization, equipped with modern multicore CPUs and, optionally, GPUs. The HPC clusters run on a Linux-based environment, as is standard in the scientific and high-performance computing scenarios due to their scalability, stability, customizability and advanced performance monitoring capabilities. Linux provides low-overhead profiling tools like *perf* and *hwloc* for hardware topology awareness. It allows fine-grained CPU affinity and NUMA-aware memory allocation, both of which are necessary for dynamic optimization strategies.

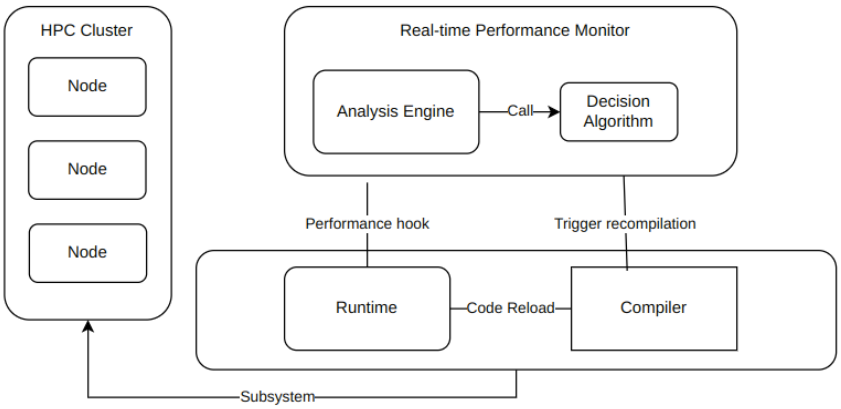

Figure 1: Architecture of Real-Time Compilation and Performance Monitoring System

B. HPC System

A High-Performance Computing (HPC) system consists of multiple interconnected nodes, with each node being equipped with multicore CPUs for high throughput execution, shared/distributed memory to hold large datasets during execution and avoid frequent memory accesses, and optionally GPUs to speedup parallel execution for SIMD workloads. HPCs operate in a Linux-based environment given its tuned performance benefits and easier low-level resource management. HPC systems are most commonly used in scientific simulations, data analysis, AI training and for any other use case where operations are performed on a massive scale on large datasets. They utilize techniques like distributed computing and vectorization to maximize throughput, to ensure performance requirements are met for resource-intensive tasks. For dynamic recompilation context, the HPC system includes a real-time performance monitor hooked into the runtime environment to capture runtime metrics. It also has a performance analysis engine to detect bottlenecks using a decision algorithm, RODA, and a compiler trigger to allow for recompilation during runtime without interrupting the program execution. This ensures that there is minimal to no manual intervention during execution.

C. HPC Workloads

The workloads of the HPC clusters typically consist of scientific and engineering applications written in low-level languages such as C, C++ and Fortran. These applications often include computationally intensive tasks in the problem domains of numerical simulations, intensive matrix transformations and data analysis pipelines. This makes them good candidates for benefitting from dynamic optimizations. Optimization decisions are more beneficial particularly in these use cases than in managed languages like Python or Java for the reason that they allow explicit control of memory through the use of pointers. Unlike JIT-based languages, these application workloads rely on static compilation, which makes adaptive recompilation very effective at improving performance dynamically based on changing runtime conditions.

D. HPC Cluster Configuration

An HPC cluster is a distributed system consisting of interconnected nodes. Each node has multiple CPUs and GPUs and hierarchical memory systems. In most cases, shared and distributed caches are used to improve memory access latency. The entire system is aware of its NUMA domains and is configured to optimize for memory accesses.

E. Performance Monitor

The performance monitor tracks system and application performance metrics. Tracked metrics include CPU utilization, memory usage and execution time. It helps with identifying bottlenecks and optimizing resource allocation. It also ensures that the system is under efficient operation. In HPC environments, performance monitoring enables real-time analysis of resource distribution and aids in tuning system parameters to maximize efficiency. Advanced metrics also provide additional information about the nature of the program being executed and its behavior, which when analyzed can be used to iron out inefficiencies.

F. Performance Metrics Collection

The real-time performance monitor is implemented using the *perf* Linux system package to collect performance metrics needed for optimization decisions. It provides low-level system performance monitors with minimal overhead using the Perf Events API, which allows directly accessing data in hardware performance counters.

The following data is gathered by the monitor during program execution:

- CPU Usage
- Memory Utilization
- Cache Miss Rate (to identify inefficient memory accesses)
- Branch Mispredictions (to detect inefficient control flows)
- Cycles Per Instruction (to determine if execution is compute-bound or memory-bound)

Performance data is captured by the monitor at regular intervals using a light-weight, asynchronous, profiling thread. This reduces overhead incurred by monitoring the performance counters. Gathered data is then used as the input to the analysis engine, which checks for potential optimizations and triggers a recompilation if found necessary for performance improvement.

G. Performance Analysis Engine

Performance data collected from the monitor is then aggregated over defined intervals to identify inefficiencies in various domains.

The analysis engine then classifies bottlenecks into categories such as:

- Memory-bound (high cache miss rate) - Candidate for cache-blocking
- Compute-bound (high CPI)- Candidate for vectorization and parallelization
- I/O-bound (high context switching)

It then uses a decision algorithm to estimate the impact of optimizations on the overall runtime of the system. If analysis indicates potential for improvement, the engine sends out a signal to the recompilation module to trigger a recompilation. This ensures that the benefits of optimization are significant enough that the overhead of recompilation does not degrade the overall performance. This results in an improvement in runtime performance. Otherwise, a recompilation is not triggered and the iteration continues.

H. Recompilation and Optimization Decision Algorithm (RODA)

The decision process of the analysis engine is driven by a decision algorithm to ensure that a recompilation is triggered only when it nets an overall improvement to the program's runtime.

The following metrics are taken into consideration:

- CPU Utilization (U_{CPU})
- Memory Bandwidth Usage (B_{mem})
- Cache Miss Ratio (C_{miss})
- Instructions Per Cycle (I_{IPC})
- Execution Time (T_{exec})

The workload is then classified as follows using Roofline Modelling:

Compute-bound -

□ = □□□□□ □□□□□□□□□ / □□□□□ □□□□□□□□□□□□, □ > □□h□□□h□□□

where,

□ = operational intensity

This likely benefits from parallelization

Memory-bound -

□□□□□ > □□□□□ □h□□□h□□□

where,

□□□□□ = observed cache miss ratio

□□□□□ □h□□□h□□□ = predefined threshold

This shows optimization potential for caching

Before triggering recompilation, the engine calculates estimated potential speedup using Amdahl's law:

$$□(□) = \frac{1}{(1-□) + \frac{□}{□} + □_{□□□□□□□□}}$$

where,
□(□) = estimated speedup with N threads
□ = parallelizable fraction of workload
□ = number of CPU threads used for parallelization
$□_{□□□□□□□□}$ = overhead incurred

Recompilation is only triggered if:

$$□ > 1 + \frac{□_{□□□□□□□□□□} + □_{□□□□□□□□} + □_{□□□□□□□□□□□□□□}}{□_{□□□□□□□□□}}$$

where,
$□_{□□□□□□□□□□}$ = overhead time for monitoring
$□_{□□□□□□□□}$ = overhead time for RODA
$□_{□□□□□□□□□□□□□□}$ = overhead time for recompilation
$□_{□□□□□□□□□}$ = estimated runtime of optimized code

RODA takes the following steps:

1. Perform Roofline Modelling to determine whether the program is compute or memory bound
2. Use Amdahl's law to determine effective speedup achieved through parallelization
3. Check if recompilation is feasible using the described relation
4. Trigger a recompilation if required, along with the appropriately identified transformation
5. Send a trigger signal to the compiler and reset monitor mode
6. Continue the iteration until maximum optimization level is reached.

I. Compiler

The compiler in an HPC system is responsible for compilation of programs written in high-level languages to efficient low-level machine code. It performs optimizations wherever possible to reduce runtime. However, most compilers used in HPC systems are limited to static compilation. Our architecture extends this by introducing dynamic compilation in response to changing runtime conditions. Static compilation generates precompiled function and code templates and includes profiling instrumentation to assist the runtime system. Whereas dynamic recompilation is facilitated by the performance analysis engine through the recompilation trigger signal. This includes an appropriately identified strategy to optimize code dynamically. The code is recompiled with given optimization parameters and then loaded onto the runtime system. Optimization strategies include loop unrolling, vectorization, parallelization and cache blocking.

J. Recompilation Module

The recompilation module performs on-the-fly code transformations based on feedback received from the analysis engine. It uses just-in-time (JIT) compilation to dynamically optimize hotspot code regions. The module applies transformations such as memory access optimizations, loop unrolling or parallelization to

the bottlenecks that are identified by RODA. Precompiled templates generated during the initial static compilation phase are used to accelerate the recompilation process. These templates include placeholders for runtime-specific optimizations such as the use of OpenMP to speedup execution. These templates can either be pre-compiled and loaded during runtime or dynamically recompiled during execution. Recompiled code is loaded into the runtime environment by replacing the corresponding memory regions. For generating the initial templates and performing dynamic code transformations, LLVM and GCC are used.

K. Runtime Environment

The runtime environment is designed to dynamically load optimized code after a recompilation is triggered. It also supports performance monitoring through the use of a real-time performance monitor. Metrics collected from the runtime system are used in the performance analysis engine to drive decisions about whether a recompilation is necessary to improve performance of the program under execution. The runtime system drives the feedback mechanism of the entire performance monitoring system to facilitate continuous and iterative performance improvements.

L. Dynamic Recompilation Trigger

Dynamic recompilation trigger primarily consists of two components: the performance analysis engine that uses RODA to determine if a recompilation is necessary, and a trigger system to send out a recompilation signal to the compiler. The decisions made by the performance analysis engine are driven by an algorithmic approach. Parameters can be modified before the monitor is started, and the engine automatically determines bottlenecks in performance based on defined thresholds. If a recompilation is needed, the appropriate transformation signal is sent out to the compiler to recompile the code. If performance degrades after recompilation, the system reverts to the previous code state using a rollback mechanism.

M. Optimization Loop

The entire runtime system is looped to allow for continuous monitoring and triggering subsequent recompilations when necessary. This loop stabilizes the program when no further optimization potential is detected while keeping the overhead to a minimum. If a previously applied compilation resulted in a performance degradation, it is reverted to the state previous to the degraded state. To prevent over-optimization, reverting between states is not done indefinitely.

4. Experimental Setup

A. Hardware

All experimental evaluations were performed on a test-bench that mimics an HPC system with following configurations:

Processor: AMD Ryzen 7 8845HS (8 cores, 16 threads)
Base Clock Frequency: 3.8 GHz
Boost Clock Frequency: Up to 5.1 GHz
Memory: 32 GB DDR5 5200 MHz RAM

L2/L3 Cache: 8 MB/16 MB

To ensure consistent results, CPU frequency scaling and other power management features were disabled during testing to prevent frequency adjustments from influencing performance. The CPU governor was set to performance mode, and the system was allowed to cool between consecutive runs to avoid thermal throttling.

Only CPU performance was evaluated since the testing was limited to parallelization decisions based on performance metrics for individual core loads and overall runtime.

B. Software

The software environment was configured with consistency of performance measurements in mind, while also ensuring compatibility with parallelization frameworks. The operating system used for all benchmarks was Arch Linux, a lightweight distribution running the Linux 6.x Arch kernel with no desktop environment to ensure headless behavior. The kernel was configured with default scheduler policies and CPU frequency scaling was explicitly disabled to maintain fixed performance states during benchmarks. The CPU governor was set to performance mode using cpupower.

For an overview of performance monitoring, resource utilization was tracked using standard Linux utilities such as htop and btop, along with a custom Python script interfacing with the /proc filesystem. The custom monitor script was written to instrument CPU utilization and individual core performance, which then used the metrics to automatically adjust the parallelization model by dynamically recompiling the Cython extension based on CPU utilization thresholds. The compiled shared object file (.so) is then loaded onto the runtime system automatically by the monitor.

All benchmarking was conducted in an isolated user environment, with no background processes running except essential system services. The system was programmed to reboot before each run to ensure ideal HPC conditions on the test bench.

C. Implementation

Cython is a programming language extension of Python. It adds static type declarations allowing integration of C and C++ code with Python code. It also allows Python code to be compiled into highly efficient C extensions, hence offering significant performance improvements in computationally intensive tasks, which are known to benefit from low-level system languages. By supporting direct calls to C libraries and facilitating parallelization through OpenMP, Cython serves as a powerful tool for optimizing performance-critical applications while also maintaining Python's flexibility and simplicity.

OpenMP (Open Multi-Processing) is an API that enables parallel programming in C, C++, and Fortran for the development of multi-threaded applications. It provides compiler directives, runtime routines and environment variables to distribute computational tasks across multiple cores. OpenMP is widely used for performance and runtime improvements in shared-memory systems through task parallelism and loop-based parallel execution.

Perf is a powerful performance analysis tool for Linux systems that provides detailed insights into system and application-level performance metrics. It utilizes hardware performance counters and software instrumentation to collect data on CPU usage, cache misses, instructions executed and other low-level metrics. Perf is commonly used in performance profiling and debugging. It helps in the identification of bottlenecks, optimization of code and understanding the runtime behavior of applications. Its versatility makes it a valuable tool for both system-wide monitoring and targeted performance analysis.

The monitor was implemented in Python 3, with performance-critical and parallelizable sections written in Cython 3. Parallelization was achieved through OpenMP using Cython's cython.parallel module. It allowed direct integration of OpenMP directives into Python extensions. The underlying C code was compiled with GCC with the -fopenmp flag explicitly set to enable OpenMP functionality during recompilation, which triggered a compilation of the Cython module. The compilation process was managed via setuptools by setting the extra_compile_args and extra_link_args parameters dynamically to enable or disable OpenMP support at runtime.

5. Benchmarking and Results

An intensive matrix transformation was used with varying sizes and iteration counts to observe how the runtime feedback affects optimization as it is scaled up. The threshold for recompilation trigger was set to be overutilization of a single core, thereby indicating a potential for parallelization. Once the threshold was reached, the monitor automatically detected a potential bottleneck and triggered a recompilation. The newly compiled code was then loaded onto the runtime system and execution was continued automatically, without any manual intervention. It was primarily observed a monotonically increasing trend in both the OpenMP and non-OpenMP versions of the code.

Matrix Size	Iterations	Recompiled Runtime	Runtime	Improvement
500x500	1000	1.277	2.342	45.5%
1000x1000	2000	3.929	7.794	49.6%
1500x1500	3000	6.452	13.113	50.8%
2000x2000	4000	10.355	21.122	50.9%
2500x2500	5000	15.533	30.882	49.7%

Table 1: Runtime Comparison of Parallelized and Non-parallelized Code

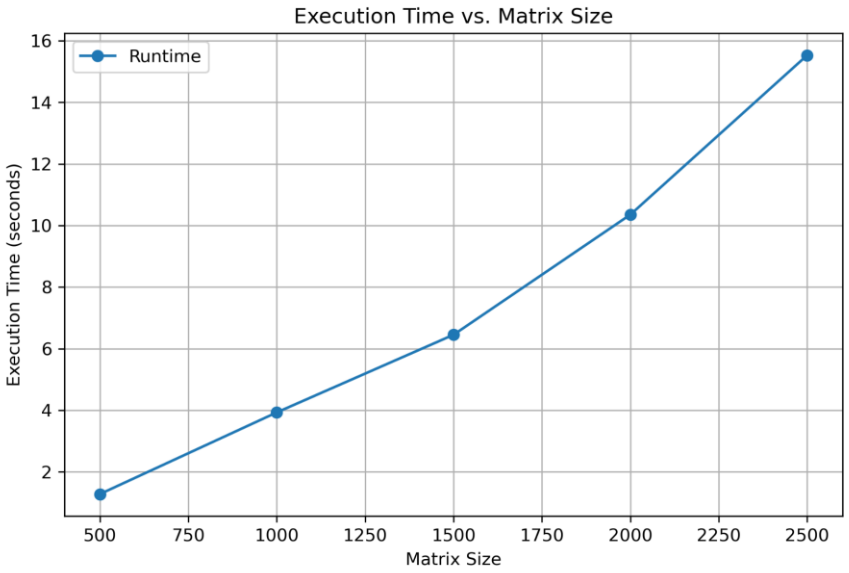

Figure 2: Runtime Metrics of OpenMP code

When compared with each other, it is observed that the non-parallelized code grows much faster in runtime than the parallelized code, hence conforming to expected behavior under parallelization. This indicates that there was a noticeable improvement in runtime performance with automatic code recompilation and parallelization.

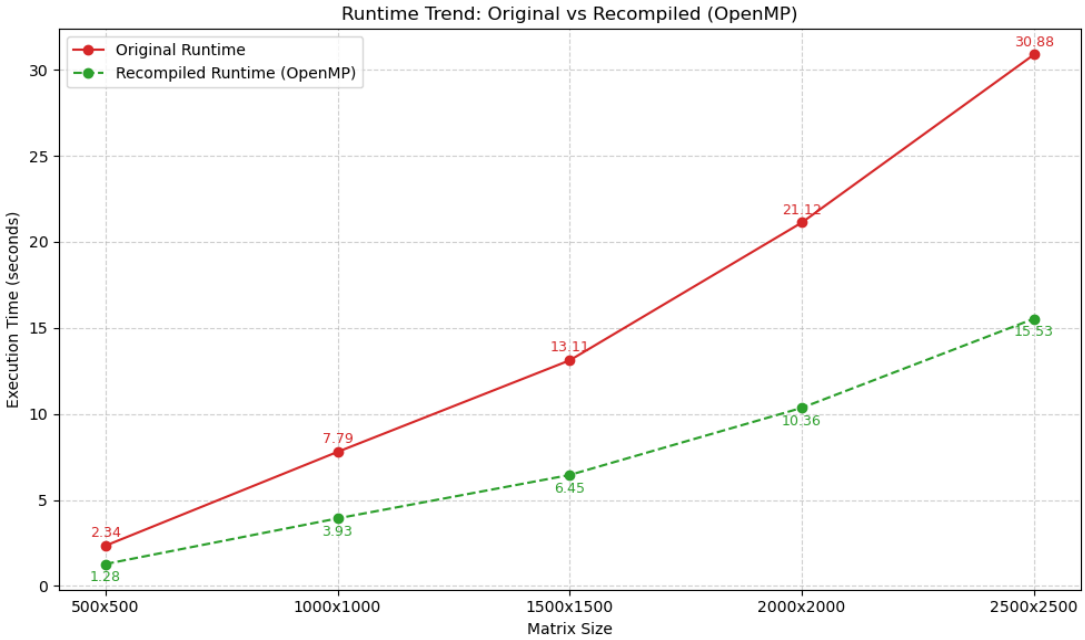

Figure 3: Comparison of OpenMP and non-OpenMP runtime

On an average, a performance improvement of about 50% can be achieved using parallelization wherever applicable. We do notice that the improvement is slightly worse when the matrix size is below a threshold, indicating that the performance overhead of spawning threads and sharing memory can

degrade overall performance. In cases like this, the monitor can be tuned to not recompile code if the performance is below the given threshold.

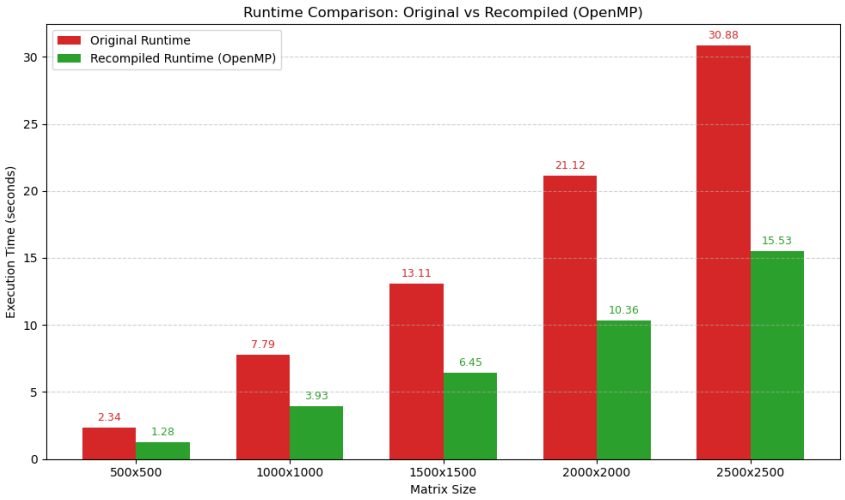

Figure 4: Performance Improvement in Parallelized Code

6. Conclusion and Future Work

This paper proposed a framework for real-time performance monitoring system and runtime feedback-driven dynamic recompilation in high performance systems. The system architecture proposed introduced a performance-aware analysis engine that is driven by a decision algorithm, RODA, to analyze program execution behavior and trigger a recompilation when found necessary to improve performance.

Recompilation triggers are accompanied by an optimizing transformation applied to the code to exploit parallelizability, cache locality and better memory access patterns. Our methodology highlights the flexibility of automatic recompilation based on predicted performance benefits, and illustrates the need for minimal programmer intervention to adjust performance during runtime. Preliminary benchmarking scoped to exploiting parallelizability of CPU heavy workloads in the form of intensive matrix transformations showed promising improvements in performance with no manual intervention. The key feature that sets the system apart is the compiler-agnostic approach to monitoring, analyzing and recompiling code, hence improving the portability and flexibility of dynamic recompiling systems. Our research lays the foundation for a new class of self-optimizing code pipelines with runtime variability in mind. Further developments for dynamic compilation based on runtime feedback will be based on multiple metrics. Including a multi-metric logic in the decision algorithm to analyze advanced metrics such as cache behavior, vectorization efficiency or thermal conditions will allow for more nuanced recompilations. Runtime behavior may be modelled using machine learning and analysis of historical data. Current design can be extended to support distributed and multi-nodes to integrate with job schedulers and MPI-based communication. Granular recompilation is another major shortcoming that needs addressing in future enhancements. It allows selected parts of the code to be recompiled, hence reducing the recompilation overhead and making dynamic and feedback driven compilation more viable for even non-HPC scenarios. Compiler integration needs further work during specific implementations in the form of plugins, to ensure portability. In addition to execution time, recompilation decisions could be driven by

energy efficiency, enabling energy savings and contributing to green computing. Most importantly, dynamic code reloading introduces security and reliability challenges that need addressing. Crash recovery and fault tolerance are areas that need more research and development to ensure safe transitions between optimized versions of code.

References

1. Hashim Sharif, Yifan Zhao, Maria Kotsifakou, Animesh Kothari, Benjamin Schreiber, Eddie Wang, Sarita Yerramilli, Nuwan Jayasena, Vikram S. Adve, Sasa Misailovic, “ApproxTuner: A Compiler and Runtime System for Adaptive Approximations”, Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP), 2021, doi: 10.1145/3437801.3446108
2. Michael P. Plezbert and Ron K. Cytron, “Fast, Effective Dynamic Compilation”, Proceedings of the ACM SIGPLAN 1997 Conference on Programming Language Design and Implementation (PLDI), 1997, doi: 10.1145/249069.231409
3. Can Özturan, Balaram Sinharoy, Boleslaw K. Szymanski, “Compiler Technology for Parallel Scientific Computation”, Scientific Programming, Volume 3, Issue 4, 1994, doi: 10.1155/1994/243495
4. Muhammad Usman, Muhammad Shafique, Jörg Henkel, “Exploiting Parallelism for Energy Efficient Source Code High Performance Computing”, Proceedings of the 2005 International Conference on High Performance Computing and Communications (HPCC), 2005, doi: 10.1007/11557654_86
5. Chun-Lung Su, Cheng-Yuan Tsui, Alvin M. Despain, “Power-Aware Compilation Techniques For High Performance Processors”, Proceedings of the 1994 IEEE Symposium on Low Power Electronics, 1994, doi: 10.1109/LPE.1994.573723
6. David F. Bacon, Susan L. Graham, Oliver J. Sharp, “Compiler Transformations for High-Performance Computing”, ACM Computing Surveys (CSUR), Volume 26, Issue 4, 1994, doi: 10.1145/197405.197406
7. Malik Khan, Protonu Basu, Gabe Rudy, Mary Hall, Chun Chen, Jacqueline Chame, “A Script-Based Autotuning Compiler System to Generate High-Performance CUDA Code”, Proceedings of the 8th International Conference on High-Performance Embedded Architectures and Compilers (HiPEAC), 2013, doi: 10.1145/2400682.2400690