

**Applying Machine Learning for Developing an
Efficient Parallel Applications Autotuning
Methodology Based on Performance Counters**

Suren Harutyunyan Gevorgyan

ADVERTIMENT. L'accés als continguts d'aquesta tesi doctoral i la seva utilització ha de respectar els drets de la persona autora. Pot ser utilitzada per a consulta o estudi personal, així com en activitats o materials d'investigació i docència en els termes establerts a l'art. 32 del Text Refós de la Llei de Propietat Intel·lectual (RDL 1/1996). Per altres utilitzacions es requereix l'autorització prèvia i expressa de la persona autora. En qualsevol cas, en la utilització dels seus continguts caldrà indicar de forma clara el nom i cognoms de la persona autora i el títol de la tesi doctoral. No s'autoritza la seva reproducció o altres formes d'explotació efectuades amb finalitats de lucre ni la seva comunicació pública des d'un lloc aliè al servei TDX. Tampoc s'autoritza la presentació del seu contingut en una finestra o marc aliè a TDX (framing). Aquesta reserva de drets afecta tant als continguts de la tesi com als seus resums i índexs.

ADVERTENCIA. El acceso a los contenidos de esta tesis doctoral y su utilización debe respetar los derechos de la persona autora. Puede ser utilizada para consulta o estudio personal, así como en actividades o materiales de investigación y docencia en los términos establecidos en el art. 32 del Texto Refundido de la Ley de Propiedad Intelectual (RDL 1/1996). Para otros usos se requiere la autorización previa y expresa de la persona autora. En cualquier caso, en la utilización de sus contenidos se deberá indicar de forma clara el nombre y apellidos de la persona autora y el título de la tesis doctoral. No se autoriza su reproducción u otras formas de explotación efectuadas con fines lucrativos ni su comunicación pública desde un sitio ajeno al servicio TDR. Tampoco se autoriza la presentación de su contenido en una ventana o marco ajeno a TDR (framing). Esta reserva de derechos afecta tanto al contenido de la tesis como a sus resúmenes e índices.

WARNING. The access to the contents of this doctoral thesis and its use must respect the rights of the author. It can be used for reference or private study, as well as research and learning activities or materials in the terms established by the 32nd article of the Spanish Consolidated Copyright Act (RDL 1/1996). Express and previous authorization of the author is required for any other uses. In any case, when using its content, full name of the author and title of the thesis must be clearly indicated. Reproduction or other forms of for profit use or public communication from outside TDX service is not allowed. Presentation of its content in a window or frame external to TDX (framing) is not authorized either. These rights affect both the content of the thesis and its abstracts and indexes.



Escola d'Enginyeria
Computer Architecture & Operating Systems Department

Applying Machine Learning for Developing an Efficient Parallel Applications Autotuning Methodology Based on Performance Counters

PhD thesis by Suren Harutyunyan Gevorgyan, for the degree of Philosophiae Doctor in Computer Science belonging to the research line of High Performance Computing, by the Universitat Autònoma de Barcelona, under the supervision of Dr. Anna Sikora and Dr. Eduardo César.

Barcelona (Spain), May, 2026



Computer Architecture & Operating Systems

Applying Machine Learning for Developing an Efficient Parallel Applications Autotuning Methodology Based on Hardware Performance Counters

Thesis submitted by Suren Harutyunyan Gevorgyan in fulfillment of the requirement for the Doctor of Philosophy degree in Computer Science by Universitat Autònoma de Barcelona. This doctoral thesis was performed in the Computer Architecture and Operative Systems department, and belongs to the High Performance Computing research line, under the supervision of Dr. Anna Sikora and Dr. Eduardo César.

Advisors signature

Advisors signature

Author signature

This work was supported by the Ministerio de Ciencia e Innovación MCIN AEI/10.13039/501100011033 under contracts PID2020-113614RB-C21 and PID2023-146193OB-I00, by the Catalan government under contract 2021 SGR 00574.

Resum

Els sistemes moderns de computació d'alt rendiment han crescut en complexitat, amb processadors que compten amb fins a 96 nuclis i arquitectures heterogènies que incorporen GPUs i acceleradors especialitzats. Aquests sistemes exposen centenars de Comptadors de Rendiment Hardware (HwPCs) que proporcionen informació detallada sobre el comportament en temps d'execució, tanmateix, la seva utilització efectiva per a l'optimització del rendiment presenta desafiaments significatius. Les limitacions hardware impedeixen el monitoratge simultani de tots els comptadors, el volum massiu de dades crea problemes d'escalabilitat, i els entorns d'execució exigeixen anàlisis ràpides per a l'ajust dinàmic. L'ajust de rendiment tradicional basat en experts s'ha tornat impracticable donats els vastos espais de paràmetres i les intrincades interaccions hardware-software. Tot i que l'Aprenentatge Automàtic (ML) ofereix promeses per automatitzar aquest procés, els enfocaments existents pateixen de portabilitat limitada, precisió de predicció inconsistent entre patrons computacionals, i manca d'automatització d'extrem a extrem que integri la identificació de regions de codi amb l'optimització de Paràmetres d'Ajust (TP).

En aquesta tesi abordem aquests problemes per a aplicacions paral·leles mitjançant el desenvolupament d'un enfocament automatitzat integrat que opera eficaçment en diverses arquitectures incloent tant CPUs com GPUs. L'enfocament consisteix en dues fases que treballen conjuntament per proporcionar optimització automàtica d'extrem a extrem.

Primer, una fase offline de reducció de HwPCs utilitza tècniques d'aprenentatge automàtic per determinar el subconjunt mínim però infor-

matiu de HwPCs necessaris tant per a la identificació de regions de codi com per a l'optimització de TP. Aquesta fase es realitza una vegada per a una arquitectura específica i analitza un conjunt de dades d'entrenament que conté etiquetes de regions de codi, mesuraments de HwPCs i valors de TP per identificar quins HwPCs són més rellevants. Mitjançant aquesta anàlisi, el nombre de HwPCs requerits per a una optimització efectiva es redueix substancialment, disminuint la sobrecàrrega de monitoratge i permetent una recollida de dades més eficient mentre es manté alta precisió en la caracterització del rendiment.

La segona fase és la fase d'execució online de l'aplicació que aplica els conjunts reduïts de HwPCs i els models entrenats identificats en la fase offline a aplicacions reals en temps d'execució. Durant l'execució, aquesta fase identifica les regions de codi en execució i determina automàticament els valors òptims de TP per a cada regió usant només el conjunt mínim de HwPCs seleccionats durant la fase offline. Aquest enfocament redueix significativament la sobrecàrrega del monitoratge de rendiment mentre manté alta precisió tant en la classificació de regions de codi com en l'optimització del rendiment, eliminant així l'esforç manual en l'anàlisi de rendiment i permetent una adaptació eficient en temps d'execució. Una característica crítica d'aquest enfocament integrat és la seva portabilitat arquitectònica, determinant automàticament els HwPCs més rellevants per a cada arquitectura objectiu durant la fase offline i assegurant una precisió de predicció consistent entre plataformes sense requerir modificacions manuals específiques de l'arquitectura. Aquest disseny agnòstic d'arquitectura s'adapta a diferents característiques arquitectòniques mentre assegura una optimització robusta de paràmetres d'ajust, permetent que el mateix enfocament s'apliqui a noves architectures sense intervenció manual.

La validació exhaustiva realitzada en múltiples plataformes hardware i càrregues de treball computacionals diverses demostra que la metodologia aborda eficaçment els desafiaments fonamentals dels entorns mod-

erns de computació paral·lela. Els resultats experimentals mostren alta precisió en la predicció de configuracions òptimes de TP en diverses regions de codi i arquitectures, amb puntuacions de precisió K-fold que freqüentment superen 0.90 i puntuacions ROC AUC consistentment per sobre de 0.99, requerint únicament entre 3 i 7 HwPCs per regió de codi de CPU, i les comparacions amb OpenTuner demostren millores en el temps d'execució de fins a 30×. La validació en sis generacions de GPU (des de Kepler fins a Ada Lovelace) confirma a més una àmplia portabilitat arquitectònica, assolint una precisió superior a 0.80 i requerint únicament entre 2 i 11 HwPCs per kernel GPU. La integració de l'anàlisi de HwPCs basat en ML amb estratègies d'optimització eficients proporciona una metodologia per a l'ajust automàtic del rendiment que elimina l'esforç d'anàlisi manual mentre manté o millora l'efectivitat d'optimització en comparació amb enfocaments tradicionals, representant un avenç en l'optimització automàtica del rendiment per a aplicacions paral·leles en entorns HPC heterogenis.

Paraules Clau

Comptadors de rendiment hardware, Reducció automàtica de dimensionalitat, Conjunts d'aprenentatge automàtic, Optimització de paràmetres d'ajust, Classificació de regions paral·leles

Resumen

Los sistemas modernos de computación de alto rendimiento han crecido en complejidad, con procesadores que cuentan con hasta 96 núcleos y arquitecturas heterogéneas que incorporan GPUs y aceleradores especializados. Estos sistemas exponen cientos de Contadores de Rendimiento Hardware (HwPCs) que proporcionan información detallada sobre el comportamiento en tiempo de ejecución; sin embargo, su utilización efectiva para la optimización del rendimiento presenta desafíos significativos. Las limitaciones hardware impiden el monitoreo simultáneo de todos los contadores, el volumen masivo de datos crea problemas de escalabilidad, y los entornos de ejecución exigen análisis rápidos para el ajuste dinámico. El ajuste de rendimiento tradicional basado en expertos se ha vuelto impracticable dados los vastos espacios de parámetros y las intrincadas interacciones hardware-software. Aunque el Aprendizaje Automático (ML) ofrece promesas para automatizar este proceso, los enfoques existentes sufren de portabilidad limitada, precisión de predicción inconsistente entre patrones computacionales y falta de automatización de extremo a extremo que integre la identificación de regiones de código con la optimización de Parámetros de Ajuste (TP).

En esta tesis abordamos estos problemas para aplicaciones paralelas mediante el desarrollo de un enfoque automatizado integrado que opera eficazmente en diversas arquitecturas incluyendo tanto CPUs como GPUs. El enfoque consiste en dos fases que trabajan conjuntamente para proporcionar optimización automática de extremo a extremo.

Primero, una fase offline de reducción de HwPCs utiliza técnicas de

aprendizaje automático para determinar el subconjunto mínimo pero informativo de HwPCs necesarios tanto para la identificación de regiones de código como para la optimización de TP. Esta fase se realiza una vez para una arquitectura específica y analiza un conjunto de datos de entrenamiento que contiene etiquetas de regiones de código, mediciones de HwPCs y valores de TP para identificar qué HwPCs son más relevantes. Mediante este análisis, el número de HwPCs requeridos para una optimización efectiva se reduce sustancialmente, disminuyendo la sobrecarga de monitoreo y permitiendo una recolección de datos más eficiente mientras se mantiene alta precisión en la caracterización del rendimiento.

La segunda fase es la fase de ejecución online de la aplicación que aplica los conjuntos reducidos de HwPCs y los modelos entrenados identificados en la fase offline a aplicaciones reales en tiempo de ejecución. Durante la ejecución, esta fase identifica las regiones de código en ejecución y determina automáticamente los valores óptimos de TP para cada región usando solo el conjunto mínimo de HwPCs seleccionados durante la fase offline. Este enfoque reduce significativamente la sobrecarga del monitoreo de rendimiento mientras mantiene alta precisión tanto en la clasificación de regiones de código como en la optimización del rendimiento, eliminando así el esfuerzo manual en el análisis de rendimiento y permitiendo una adaptación eficiente en tiempo de ejecución. Una característica crítica de este enfoque integrado es su portabilidad arquitectónica, determinando automáticamente los HwPCs más relevantes para cada arquitectura objetivo durante la fase offline y asegurando una precisión de predicción consistente entre plataformas sin requerir modificaciones manuales específicas de la arquitectura. Este diseño agnóstico de arquitectura se adapta a diferentes características arquitectónicas mientras asegura una optimización robusta de parámetros de ajuste, permitiendo que el mismo enfoque se aplique a nuevas arquitecturas sin intervención manual.

La validación exhaustiva realizada en múltiples plataformas hardware y

cargas de trabajo computacionales diversas demuestra que la metodología aborda eficazmente los desafíos fundamentales de los entornos modernos de computación paralela. Los resultados experimentales muestran alta precisión en la predicción de configuraciones óptimas de TP en diversas regiones de código y arquitecturas, con puntuaciones de precisión K-fold que frecuentemente superan 0.90 y puntuaciones ROC AUC consistentemente por encima de 0.99, requiriendo únicamente entre 3 y 7 HwPCs por región de código de CPU y las comparaciones con OpenTuner demuestran mejoras en el tiempo de ejecución de hasta 30×. La validación en seis generaciones de GPU (desde Kepler hasta Ada Lovelace) confirma además una amplia portabilidad arquitectónica, alcanzando una precisión superior a 0.80 y requiriendo únicamente entre 2 y 11 HwPCs por kernel GPU. La integración del análisis de HwPCs basado en ML con estrategias de optimización eficientes proporciona una metodología para el ajuste automático del rendimiento que elimina el esfuerzo de análisis manual mientras mantiene o mejora la efectividad de optimización en comparación con enfoques tradicionales, representando un avance en la optimización automática del rendimiento para aplicaciones paralelas en entornos HPC heterogéneos.

Palabras Clave

Contadores de rendimiento hardware, Reducción automática de dimensionalidad, Conjuntos de aprendizaje automático, Optimización de parámetros de ajuste, Clasificación de regiones paralelas

Summary

Modern High Performance Computing (HPC) systems have grown increasingly complex, with processors featuring up to 96 cores and heterogeneous architectures incorporating GPUs and specialized accelerators. These systems expose hundreds of Hardware Performance Counters (HwPCs) that provide detailed insights into runtime behavior; yet their effective utilization for performance optimization presents significant challenges. Hardware limitations prevent simultaneous monitoring of all counters, the massive data volume creates scalability issues, and runtime environments demand rapid analysis for dynamic tuning. Traditional expert-driven performance tuning has become impractical given the vast parameter spaces and intricate hardware-software interactions. While Machine Learning (ML) offers promise for automating this process, existing approaches suffer from limited portability, inconsistent prediction accuracy across computational patterns, and lack of end-to-end automation integrating code-region identification with Tuning Parameter (TP) optimization.

In this thesis we tackle these problems for parallel applications through the development of an integrated automated approach that operates effectively across diverse architectures including both CPUs and GPUs. The approach consists of two phases that work together to provide end-to-end automatic optimization.

First, an offline HwPC reduction phase uses ML techniques to determine the minimal yet informative subset of HwPCs necessary for both code-region identification and TP optimization. This phase is performed once for a specific architecture and analyzes a training dataset

containing code-region labels, HwPC measurements, and TP values to identify which HwPCs are most relevant. Through this analysis, the number of HwPCs required for effective optimization is substantially reduced, decreasing monitoring overhead and enabling more efficient data collection while maintaining high accuracy in performance characterization.

The second phase is the online application execution phase that applies the reduced HwPC sets and trained models identified in the offline phase to real applications at runtime. During execution, this phase identifies executing code-regions and automatically determines optimal TP values for each region using only the minimal set of HwPCs selected during the offline phase. This approach significantly reduces the overhead of performance monitoring while maintaining high accuracy in both code-region classification and performance optimization, thereby removing manual effort in performance analysis and enabling efficient runtime adaptation. A critical characteristic of this integrated approach is its architectural portability, automatically determining the most relevant HwPCs for each target architecture during the offline phase and ensuring consistent prediction accuracy across platforms without requiring manual architecture-specific modifications. This architecture-agnostic design adapts to different architectural characteristics while ensuring robust tuning parameter optimization, enabling the same approach to be applied to new architectures without manual intervention.

Comprehensive validation conducted across multiple hardware platforms and diverse computational workloads demonstrates that the methodology effectively addresses the fundamental challenges of modern parallel computing environments. Experimental results show high accuracy in predicting optimal TP configurations across diverse code-regions and architectures, with K-fold accuracy scores frequently exceeding 0.90 and ROC AUC scores consistently above 0.99, while requiring only 3–7 HwPCs per CPU code-region, and benchmarks against

OpenTuner demonstrate execution time improvements of up to $30\times$. Validation across six GPU generations (Kepler through Ada Lovelace) further confirms broad architectural portability, achieving above 0.80 accuracy while requiring only 2–11 HwPCs per GPU kernel. The integration of ML-based HwPC analysis with efficient optimization strategies provides a methodology for automated performance tuning that eliminates manual analysis effort while maintaining or improving optimization effectiveness compared to traditional approaches, representing an advancement in automated performance optimization for parallel applications in heterogeneous HPC environments.

Keywords

Hardware performance counters, Automatic dimension reduction, Machine learning ensembles, Tuning parameter optimization, Parallel region classification

Acknowledgements

I would like to express my gratitude to all the people who have contributed to making this thesis possible.

First and foremost, I want to thank my advisors Anna Sikora and Eduardo César for their continuous guidance and support throughout these years. Their ability to steer the research in the right direction and propose new ideas has been essential to the progress and quality of this work.

I am also grateful to Andreu Moreno, Josep Jorba and Joan Sorribes, whose participation in our regular meetings has been very enriching. Their updates on their own research, combined with the suggestions and warnings about potential pitfalls they generously shared, have been of great help along the way. I also want to give special thanks to Jordi Alcaraz, whose assistance during the early stages of the PhD helped lay a solid foundation for everything that followed.

My thanks also go to everyone in our department (CAOS), including the teaching staff who offered their help and encouragement in many different ways, as well as the administrative and support members, particularly Dani Ruiz and Gemma Roque, whose work keeps everything running behind the scenes.

I would also like to acknowledge my colleagues, both those who have already completed their doctorates and those still working towards theirs. Sharing the day-to-day experience of research life has made this journey more rewarding. I wish those still on their way all the best in reaching the finish line.

On the international side, I have had the privilege of collaborating with Jiří Filipovič from Masaryk University, whose ideas and insights have greatly enriched this work. I am equally thankful for the opportunity he provided to visit Masaryk University, and for his help in making my stay in Brno as smooth as possible. I would also like to thank Ali Jannesari and Akash Dutta from Iowa State University for a stimulating and productive collaboration that has been a source of new perspectives throughout these years.

I would like to express my gratitude to Isabel Serra and David Romero for their guidance and support before the start of this PhD. It played a key role in shaping my path and laying the foundations for this work.

Finally, I would like to thank Carles Grabuleda and Elisa Alòs for their mentorship and encouragement, which have been a source of strength and motivation at key moments in this journey.

Contents

Contents	xv
List of Figures	xix
List of Tables	xxiii
1 Introduction	1
1.1 Motivation	1
1.2 Objectives	6
1.3 Main Contributions	7
1.4 Thesis Organization	9
2 Related Work	11
2.1 Analytical Performance Modeling	12
2.1.1 MATE	13
2.1.2 Periscope	14
2.1.3 Score-P	14
2.1.4 Scalasca	16
2.1.5 Extra-P	17
2.2 Autotuning with Search in the Tunable Parameters' Space	18
2.2.1 OpenTuner	18
2.2.2 Kernel Tuning Toolkit	19
2.2.3 Kernel Tuner	20
2.3 Tuning Methodologies Based on ML	21

CONTENTS

2.3.1	READEX	21
2.3.2	BLISS	22
2.3.3	Predictive Modeling for Thread Optimization in OpenMP- based Parallelization Using ML	23
2.3.4	Artemis	24
2.3.5	Learning to Select Scheduling Algorithms in OpenMP	25
2.3.6	Automated Algorithm Design for Autotuning Optimizers . .	26
2.4	Tuning Methodologies Based on ML and Hardware Performance Counters	26
2.4.1	Hardware Performance Counter Reliability	27
2.4.2	Hardware Performance Counters for GPU Autotuning	28
2.4.3	APARF	29
2.4.4	Piecewise Holistic Autotuning with CERE	30
2.4.5	Pattern-based Autotuning of OpenMP Loops Using Graph Neural Networks	31
2.5	Comparison of the presented approaches	32
3	Background	39
3.1	Previous Work	39
3.2	Dataset Construction	41
3.2.1	Preset Events and Compatibilities	42
3.2.2	Problem Sizes	43
3.3	Machine Learning	44
3.3.1	Logistic Regression with Regularization	45
3.3.2	Tree Based Models	47
3.3.3	TabNet	51
3.4	Ensemble Methodology	54
3.4.1	Ensemble Evaluation Metrics	55
3.5	Learning Outcomes	58
4	Methodology For Hardware Performance Counters...	61
4.1	HwPC Data and Machine Learning	63
4.1.1	HwPCs and Performance Optimization	63

4.1.2	Ensemble Algorithms	64
4.2	Dataset Construction	67
4.2.1	Hardware Architecture	68
4.2.2	Preset Events per System	68
4.2.3	Problem Size Calculation	71
4.2.4	OpenMP Tuning Parameters	73
4.2.5	Execution Methodology	76
4.3	Application Performance Tuning using Machine Learning	78
4.4	HwPC Reduction and Model Training	80
4.4.1	Ensemble Training and Validation	81
4.4.2	Weight Extraction	82
4.4.3	HwPC Ranking and Reduction	83
4.5	Code-Region Identification and Application Execution Phase	85
5	Evaluation of the Proposed Methodology	87
5.1	Comparison with PCA and Correlation-Based Methodology	88
5.2	Identification and Tuning of OpenMP Regions with a Comprehensive Dataset	89
5.2.1	Evaluation Dataset	90
5.2.2	Code-Region Identification	91
5.2.3	Tuning Parameter: Number of Threads	94
5.2.4	Tuning Parameter: Thread Affinity Policy	100
5.2.5	Tuning Parameter: Scheduling Policy and Chunk Size	104
5.3	Application to the NAS Parallel Benchamarks	107
5.3.1	Evaluation Dataset	107
5.3.2	Code-Region Identification	109
5.3.3	Tuning Parameter: Number of Optimal Threads	109
5.3.4	Tuning Parameter: Thread Affinity Policy	113
5.4	Applying the Methodology to GPUs	116
5.4.1	Evaluation Dataset	116
5.4.2	Tuning Parameter Evaluation Results	118
5.5	Discussion of the Evaluation Results	122

CONTENTS

6	Conclusions and Future Work	131
6.1	Conclusions	131
6.2	Future Work	136
	References	139

List of Figures

1.1	General flow of the proposed methodology.	9
3.1	Bagging process.	49
3.2	Boosting process.	50
3.3	TabNet model architecture.	53
3.4	Theoretical model ensembling process.	55
4.1	Detailed flow of the proposed methodology for automatic tuning of parallel applications.	79
4.2	HwPCs reduction with model ensembles.	81
4.3	Step 2: HwPC importance per model and weighted average.	83
4.4	Step 3: HwPC reduction considering the knee point.	85
4.5	Code-region identification and TP optimization with model ensembles.	86
5.1	Evaluation results of the ensemble for the prediction of the number of threads TP minimizing $P_i(X)$	96
5.2	Evaluation results of the ensemble for the prediction of the number of threads TP minimizing the execution time.	98
5.3	Comparison of the optimal (Opt.), ensemble minimizing $P_i(X)$ (Perf.), and OpenTuner (OpenT.) mean performance indices.	99
5.4	Comparison of the optimal (Opt.), ensemble minimizing $P_i(X)$ (Perf.), ensemble minimizing execution time (Time), and OpenTuner (OpenT.) mean speedups.	100

LIST OF FIGURES

5.5	Comparison of the execution times (logarithmic scale) required for computing the number of threads: ensemble minimizing $P_i(X)$ (Perf.), ensemble minimizing execution time (Time), and OpenTuner (OpenT.).	100
5.6	Evaluation results of the ensemble for the prediction of the affinity TP.	102
5.7	Evaluation results of OpenTuner for the prediction of the affinity TP.	103
5.8	Comparison of the execution times (logarithmic scale) required for computing affinity: ensemble and OpenTuner.	106
5.9	Evaluation results of the ensemble for the code-region identification (NAS benchmarks).	110
5.10	Comparison of the optimal (Opt.), ensemble minimizing $P_i(X)$ (Perf), and OpenTuner (OpenT.) mean performance indices (NAS benchmarks).	111
5.11	Comparison of the optimal (Opt.), ensemble minimizing $P_i(X)$ (Perf), ensemble minimizing execution time (Time), and OpenTuner (OpenT.) mean speedups (NAS benchmarks).	112
5.12	Comparison of the execution times (logarithmic scale) required for computing the number of threads: ensemble minimizing $P_i(X)$ (Perf), ensemble minimizing execution time (Time), and OpenTuner (OpenT) (NAS benchmarks).	112
5.13	Evaluation results of the ensemble for the prediction of the affinity TP (NAS benchmarks).	114
5.14	Evaluation results of OpenTuner for the prediction of the affinity TP (NAS benchmarks).	115
5.15	Comparison of the execution times (logarithmic scale) required for computing affinity: ensemble and OpenTuner (NAS benchmarks). .	115
5.16	Evaluation results of the ensemble for the HwPC reduction and kerne identification for GPUs.	120
5.17	Averaged evaluation results of the ensemble for the prediction of the TPs for the Kepler GPU.	123
5.18	Averaged evaluation results of the ensemble for the prediction of the TPs for the Maxwell GPU.	123

5.19	Averaged evaluation results of the ensemble for the prediction of the TPs for the Pascal GPU.	124
5.20	Averaged evaluation results of the ensemble for the prediction of the TPs for the Turing GPU.	124
5.21	Averaged evaluation results of the ensemble for the prediction of the TPs for the Ampere GPU.	125
5.22	Averaged evaluation results of the ensemble for the prediction of the TPs for the Ada Lovelace GPU.	125

List of Tables

4.1	Hardware architecture details.	69
4.2	Preset HwPCs available on the Xeon E5-4620 platform.	70
4.3	Compatible PAPI preset event groups for the Intel E5-4620 system.	71
4.4	Problem sizes calculated for different memory levels in Xeon E5-4620.	73
4.5	OpenMP TPs and their configuration values for the Xeon E5-4620 platform.	76
4.6	Number of the execution configurations for a specific platform (Xeon E5-4620).	77
5.1	Preset HwPCs available on the Xeon E5645 platform.	88
5.2	Reduced HwPCs for code-region identification.	89
5.3	Evaluation results for Xeon E5645.	89
5.4	Code-regions grouped by benchmark suite and computational characteristics.	92
5.5	Preset HwPCs available on the Xeon E5-4620 platform.	92
5.6	Evaluation results for code-region identification for the expanded benchmark on the Xeon E5-4620.	93
5.7	Reduced HwPCs for code-region identification.	94
5.8	Reduced HwPCs for 2D4PStencil_E code-region's number of threads optimization.	94
5.9	HwPCs used by each code-region for number of threads optimization (grouped by preset event type).	95
5.10	Full evaluation results of the ensemble for the prediction of the number of threads TP minimizing $P_i(X)$	97

LIST OF TABLES

5.11	Full evaluation results of the ensemble for the prediction of the number of threads TP minimizing the execution time.	98
5.12	Full comparison of the optimal, ensemble minimizing $P_i(X)$ (Perf), and OpenTuner mean performance indices.	101
5.13	Full comparison of the optimal, ensemble minimizing $P_i(X)$ (Perf.), ensemble minimizing execution time (Time), and OpenTuner mean speedups.	102
5.14	Full comparison of the execution times required for computing the number of threads: ensemble minimizing $P_i(X)$ (Perf.), ensemble minimizing execution time (Time), and OpenTuner.	103
5.15	Full evaluation results of the ensemble for the prediction of the affinity TP.	104
5.16	Full evaluation results of OpenTuner for the prediction of the affinity TP.	105
5.17	Full comparison of the execution times (logarithmic scale) required for computing affinity: ensemble and OpenTuner.	106
5.18	Scheduling policy and chunk size parameter evaluation results for the expanded benchmark on the Xeon E5-4620.	107
5.19	Scheduling policy and chunk size parameter OpenTuner evaluation results for the expanded benchmark on the Xeon E5-4620.	107
5.20	Comparison of the ensemble, and OpenTuner mean execution times for the expanded benchmark on the Xeon E5-4620. Focus on scheduling policy and chunk size.	107
5.21	NAS Parallel Benchmark code-regions grouped by benchmark suite.	109
5.22	Full evaluation results of the ensemble for the code-region identification (NAS benchmarks).	110
5.23	Full comparison of the optimal, ensemble minimizing $P_i(X)$ (Perf), and OpenTuner mean performance indices (NAS benchmarks). . . .	111
5.24	Full comparison of the optimal, ensemble minimizing $P_i(X)$ (Perf), ensemble minimizing execution time (Time), and OpenTuner mean speedups (NAS benchmarks).	112

5.25	Full comparison of the execution times required for computing the number of threads: ensemble minimizing $P_i(X)$ (Perf), ensemble minimizing execution time (Time), and OpenTuner (NAS benchmarks).	113
5.26	Full evaluation results of the ensemble for the prediction of the affinity TP (NAS benchmarks).	113
5.27	Full evaluation results of OpenTuner for the prediction of the affinity TP (NAS benchmarks).	114
5.28	Full comparison of the execution times required for computing affinity: ensemble and OpenTuner (NAS benchmarks).	116
5.29	GPU kernels grouped by tested architecture.	117
5.30	Common TPs by kernel. Tile size is ticked when it can be configured differently than Work-Group size.	117
5.31	Reduced HwPCs for code-region identification across GPU architectures.	119
5.32	HwPC reduction and kernel identification evaluation results for GPUs.	120
5.33	Kernel TP evaluation average results on the GPUs.	126
5.34	Reduced HwPCs for predicting WPTX TP for the Convolution kernel across GPU architectures.	127

Chapter 1

Introduction

This chapter introduces the problems we have tackled in this thesis focused on the field of automatic performance optimization of parallel applications using Machine Learning (ML) techniques and Hardware Performance Counters (HwPCs). First, the motivation behind this work is discussed, followed by the objectives we plan to accomplish in this thesis. Afterwards, the main contributions of this thesis are explained, and, finally, the organization of this work is presented.

1.1 Motivation

Nowadays, computational demands are growing exponentially driven by increasingly complex simulations, artificial intelligence workloads, and data-intensive applications across diverse fields. To meet these requirements, more sophisticated systems with enhanced computational capabilities are being developed in High Performance Computing (HPC). Consequently, a continuous cycle appears to be generated between the development of more demanding applications and advances in computational power. This increase in demand is satisfied through significant expansions in system resources, such as increasing the number of cores per processor and deploying systems with multiple processors and accelerators.

For instance, while Intel’s 2005-era Paxville DP (Xeon 7000 series) and 2006-era

Dempsey (Xeon 5000 series) processors both featured 2 cores and 4 threads for dual-socket servers, current Intel Xeon Scalable Processors (Sapphire Rapids/Emerald Rapids lineup) can offer up to 64 cores with 128 threads (e.g., Platinum 8592+ [34]). Similarly, AMD’s EPYC 9004 Series (codenamed Genoa) processors provide configurations with up to 96 cores and 192 threads (e.g. EPYC 9654 [2]). Additionally, due to power consumption constraints and thermal limitations, systems are becoming increasingly heterogeneous, incorporating technologies such as dynamic voltage and frequency scaling, hybrid architectures with performance and efficiency cores, and specialized accelerators including GPUs, FPGAs, and other domain-specific processors.

Such architectures provide parallel applications with vast opportunities to exploit thread-level parallelism, vectorization, and hardware specialization for performance gains. To effectively utilize these complex systems and extract maximum performance from parallel applications, performance tuning becomes crucial. However, the growing complexity of modern HPC architectures makes manual performance optimization increasingly challenging and time-consuming. Traditional approaches that rely on expert knowledge and manual analysis become impractical when dealing with the vast parameter spaces and intricate interactions between hardware components and software configurations. This complexity is further compounded by the heterogeneous nature of modern systems, where optimal configurations may vary significantly between different architectures and workloads.

For instance, recent Intel and AMD CPUs expose over 200 architectural, and microarchitectural events, while NVIDIA GPUs provide access to several hundred specialized Hardware Performance Counters (HwPCs). These HwPCs offer unprecedented visibility into the runtime behavior of applications, enabling detailed analysis of performance bottlenecks and optimization opportunities.

However, the effective use of HwPCs for parallel application tuning presents several significant challenges that must be addressed:

1. **Monitoring Limitations:** Hardware factors, such as the limited number of special-purpose registers, restrict simultaneous access to HwPCs. Two ap-

proaches address this limitation: multiple application executions or HwPC multiplexing. Multiple executions achieve high accuracy by measuring different HwPC groups in separate runs, but this multiplies execution time and is infeasible for long-running applications or dynamic tuning scenarios. HwPC multiplexing timeshares register usage among multiple events, requiring only a single execution. However, this introduces overhead from HwPC swapping and reduces precision, as final values must be estimated from partial measurements across time intervals.

2. **Scalability and Data Volume Constraints:** The complexity of modern HPC systems, featuring processors with hundreds of cores and heterogeneous accelerators, has led to an explosion in available HwPCs. The volume of HwPC data, especially from GPUs, with hundreds of associated HwPCs, implies serious scalability challenges. Manual analysis becomes impractical when dealing with hundreds of HwPCs across thousands of processing elements, necessitating automated approaches to efficiently process this data.
3. **Runtime Decision Requirements:** HPC environments require rapid analysis and decision-making, where HwPC data proves valuable for enhancing dynamic tuning processes. The most critical aspect involves developing efficient performance models that accurately represent application behavior by incorporating instrumentation-derived HwPCs, enabling real-time optimization decisions.
4. **Manual Analysis Complexity:** The intricate relationships between HwPCs and performance outcomes make it difficult to identify performance bottlenecks and optimization opportunities manually. The multidimensional nature of the optimization problem, involving numerous interdependent hardware and software parameters, exceeds human capacity for systematic exploration. Furthermore, the architectural diversity across modern heterogeneous platforms requires specialized knowledge that varies by system, limiting the portability of manual analysis strategies. Sophisticated analytical techniques and extensive expertise in hardware microarchitecture and application behavior are essential, yet the scale and complexity of contem-

1. Introduction

porary HPC environments make analytical approaches—whether manual or automated—increasingly impractical.

All these challenges are particularly acute in the context of parallel application tuning, which has made traditional expert-driven performance modeling increasingly difficult and time-consuming. Performance tuning in these environments requires analyzing the intricate relationships between hardware behavior (captured by HwPCs), application characteristics, and tunable parameters (TPs) such as number of threads, scheduling policies, and memory allocation strategies. The manual creation of performance models that accurately capture these relationships across different architectures and workloads has become impractical, necessitating automated approaches that can systematically analyze HwPC data to predict optimal TP configurations.

Concurrently, the field of ML has experienced remarkable growth and success in handling complex, high-dimensional data analysis problems. ML techniques have demonstrated exceptional capability in identifying patterns, making predictions, and automating decision-making processes across different scientific and industrial fields. Given these strengths, ML approaches offer a promising solution to the challenges of performance optimization using HwPCs:

- **Efficient HwPC Selection:** Identifying the most informative subset of HwPCs that can both identify code-regions and optimize Tuning Parameters (TPs), thereby reducing monitoring overhead.
- **Automated Pattern Recognition:** Detecting patterns for accurate code-region characterization.
- **Predictive Tuning:** Learning optimal TP configurations from performance data.
- **Cross-Architecture Portability:** Adapting to diverse hardware like CPUs and GPUs.

However, applying ML to HwPC-based performance optimization comes with significant challenges. First, the ML model architecture must be carefully chosen

1. Introduction

to capture the relationships between HwPC measurements and performance outcomes. Second, the principle of *garbage in, garbage out* applies—an ML model is only as good as its training data. For HwPC-based performance tuning, this means the model can only learn meaningful patterns if the input features (HwPCs) capture relevant performance characteristics, and the training dataset represents the diverse scenarios the ML model will encounter in practice.

Previous research has demonstrated promising results in leveraging ML methodologies for HwPC-based parallel application tuning, including work on identifying code-regions and constructing balanced datasets to predict optimal number of threads for OpenMP code-regions. However, significant limitations remain when scaling these approaches.

- **Limited Architectural Portability:** Validation primarily on single hardware platforms restricts applicability across diverse architectures.
- **Inconsistent Prediction Accuracy:** Current approaches show accuracy variations when predicting optimal configurations across different computational patterns.
- **Lack of End-to-End Automation:** There is a lack of automated and portable methodologies that can both identify code-regions and perform Tuning Parameter optimization across diverse code-regions and architectures.

Therefore, a critical challenge remains: developing an automated methodology that can leverage ML techniques and HwPCs to seamlessly identify parallel code-regions—distinct computational blocks within an application that can be independently optimized—and optimize TPs across diverse architectures. Addressing this challenge requires approaches that can automatically identify relevant performance characteristics, select appropriate optimization strategies, and predict optimal configurations with minimal overhead and maximum accuracy. Finally, these approaches should provide portable solutions that generalize across different computational patterns and hardware platforms.

1.2 Objectives

The main objective of this thesis is to research and develop automated methodologies that leverage ML techniques and HwPCs for efficient performance optimization of parallel applications. The approach aims to eliminate the need for manual analysis while providing a portable solution that works effectively across diverse hardware architectures and application types. To achieve this overarching goal, several specific objectives need to be fulfilled:

1. **Representative Dataset Generation:** Develop a methodology for creating representative and balanced datasets that effectively capture the diverse behaviors exhibited by parallel applications across different scenarios, problem sizes, and hardware configurations. This includes addressing challenges related to data imbalance, ensuring adequate coverage of the parameter space, and maintaining the generalizability of trained models across different contexts while enabling accurate tuning capabilities.
2. **Effective HwPC Utilization and Analysis:** Develop an automated methodology for effective utilization of HwPCs in ML-based optimization systems. This involves addressing the challenges of HwPC selection, data collection overhead, and scalability issues associated with analyzing large volumes of performance data. The objective is to create an automated methodology that can identify the most informative subset of HwPCs while minimizing measurement overhead and maintaining high accuracy in performance characterization. Additionally, this requires developing a performance model capable of rapidly analyzing HwPC data and making optimization decisions with minimal latency, enabling dynamic tuning that responds effectively to changing application behavior.
3. **Portable Optimization Methodologies:** Develop a portable optimization methodology that can operate effectively across diverse hardware architectures, including both CPUs and GPUs. This requires creating a methodology that automatically adapts to different architectural characteristics while ensuring robust Tuning Parameter optimization across platforms, such that

both HwPC selection and optimization strategies can be applied to new architectures without manual intervention or architecture-specific adjustments.

4. **End-to-End Automation:** Provide a comprehensive automation that seamlessly integrates code-region identification and Tuning Parameter optimization. The objective is to create a system that can automatically analyze parallel applications, identify optimization opportunities, and predict optimal configurations without requiring a manual intervention or expert knowledge, addressing the current absence of such automated methodologies.

1.3 Main Contributions

This thesis demonstrates that automated methodologies leveraging ML techniques and HwPCs can effectively optimize parallel applications across diverse scenarios and architectures. This thesis presents the following main contributions:

- **ML-Based Automatic HwPC Reduction:** An automatic and accurate methodology that can determine a reduced subset of HwPCs for code-region identification and Tuning Parameter (TP) optimization, regardless of the total number of available HwPCs or code-regions. This contribution has been published in [29]:

Efficient code-region Characterization Through Automatic Performance Counters Reduction Using Machine Learning Techniques. Euro-Par (1) 2024: 18-32. Suren Harutyunyan Gevorgyan, Eduardo César, Anna Sikora, Jiří Filipovič, Akash Dutta, Ali Jannesari, Jordi Alcaraz

- **ML-Based Automatic Code-Region Optimization:** An advanced automatic methodology that accurately identifies code-regions and automates Tuning Parameter (TP) optimization using minimal yet informative sets of Hardware Performance Counters (HwPCs). The methodology operates in two distinct phases: (1) an off-line **HwPC Reduction Phase** performed once for a specific architecture, which processes a training dataset containing code-region labels, HwPC measurements, and TP values to separately

identify the minimal HwPCs needed for code-region identification and those needed for TP optimization; and (2) an on-line **Application Execution Phase** that applies the reduced HwPC sets and trained models to identify executing code-regions at runtime and automatically determine optimal TP values for each region. This two-phase approach significantly reduces the overhead of performance monitoring while maintaining high accuracy in both code-region classification and performance optimization, thereby removing manual effort in performance analysis and enabling efficient runtime adaptation. This contribution has been published in [30]:

Automatic Tuning based on Hardware Performance Counters and Machine Learning. Future Generation Computer Systems Special Issue 2026. Suren Harutyunyan Gevorgyan, Eduardo César, Anna Sikora, Jiří Filipovič, Jordi Alcaraz.

- **Portable Optimization Methodology:** An automatic methodology that is effective across diverse architectures, including CPUs and GPUs. The methodology automatically determines the most relevant HwPCs for each target architecture, ensuring consistent prediction accuracy across platforms without requiring manual architecture-specific modifications.

The methodology leverages Machine Learning to address two fundamental challenges in performance optimization: identifying which code-region is executing and determining optimal Tuning Parameter (TP) configurations. The methodology is built on the insight that different HwPCs are relevant for different optimization tasks. The HwPCs most useful for code-region identification may differ from those that best predict optimal TP values.

By recognizing this distinction, the methodology employs separate HwPC subsets that maximize predictive accuracy while minimizing monitoring overhead. The methodology operates in two phases, as illustrated in Figure 1.1.

An offline **HwPC Reduction Phase** analyzes a dataset to perform a reduction and identify which are the minimal sets of HwPCs that are most informative for code-region identification and TP optimization. This phase builds classification

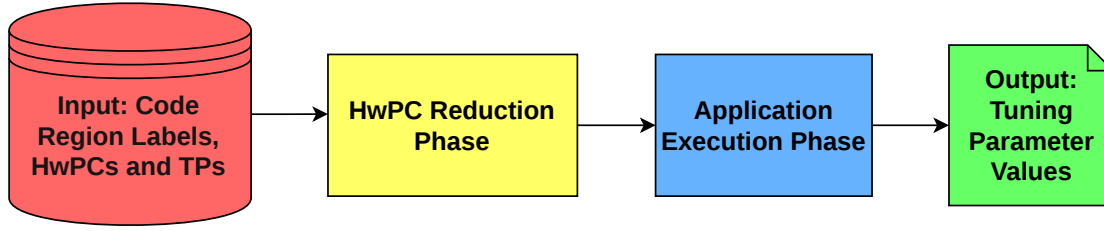


Figure 1.1: General flow of the proposed methodology.

and prediction models using an ensemble of ML algorithms.

A subsequent **Application Execution Phase** applies these trained models at runtime, using the reduced HwPC sets to identify code-regions and optimize TPs automatically. This strategy enables efficient performance tuning by monitoring only minimal HwPC subsets during application execution while maintaining high model accuracy.

1.4 Thesis Organization

This thesis is organized into the following chapters:

Chapter 2 (Related Work) examines existing research related to performance tuning and ML applications for performance analysis and optimization, providing the context for this work.

Chapter 3 (Background) outlines the fundamental concepts, methodologies, and approaches employed in this research to achieve the previously stated objectives.

Chapter 4 (Methodology For Hardware Performance Counters Reduction and Automatic Tuning using Machine Learning) presents the specific steps performed to build the representative dataset employed to validate the proposed methodology, and details the criteria for selecting the algorithms forming the ensemble methodology. Next, it introduces the methodology employed to reduce the HwPCs and obtain minimal sets of HwPCs for effective code-region identification and TP optimization. Finally, it explains how the ensemble methodology and

the reduced sets of HwPCs are integrated to optimize TPs across diverse parallel applications and architectures.

Chapter 5 (Evaluation of the Proposed Methodology) presents the comprehensive evaluation of the ensemble methodology for HwPC reduction, code-region identification, and TP optimization. The evaluation begins with OpenMP code-regions on CPUs, first validating the methodology’s reliability through comparative analysis, then assessing performance on datasets containing code-regions known to the models. It then demonstrates robustness and generalization capabilities on previously unseen code-regions from the NAS parallel benchmarks, and extends the analysis to six GPU architectures across five GPU datasets.

Chapter 6 (Conclusions and Future Work) provides a comprehensive summary of the thesis research and the results achieved through the application of the described methodology. Additionally, it identifies potential avenues for future research aimed at enhancing both the methodologies themselves and the performance models developed using the techniques detailed in this thesis.

Chapter 2

Related Work

This chapter reviews the state-of-the-art in performance modeling of parallel applications, categorized into four main approaches: analytical methodologies, autotuning with search in the tunable parameters' space, methodologies based on ML, and methodologies based on ML and HwPCs.

First, we examine analytical performance modeling approaches. These methods rely on expert knowledge to establish relationships between metrics and performance through explicit performance models or predefined patterns. Analytical approaches include dynamic performance analysis tools that use expert-defined models and plugin-based architectures to identify performance bottlenecks through instrumentation and runtime monitoring. These techniques form the foundation of performance optimization, though they often require significant domain expertise.

Second, we discuss autotuning tools that employ search algorithms to explore the parameter space, where autotuning refers to the automatic adjustment of code to specific hardware and input characteristics by searching through tuning spaces—the cross product of performance-relevant source code parameters. Rather than relying on predefined performance models or ML, these tools navigate combinations of TPs using various search strategies—from evolutionary algorithms to grid search—to find optimal configurations. While effective, these approaches may require exhaustive or near-exhaustive exploration of large parameter spaces.

2. Related Work

Third, we explore methodologies based on ML that do not explicitly leverage HwPCs. These approaches use ML models to predict optimal configurations or make runtime decisions based on various application features, such as code characteristics, execution patterns, or system parameters. By learning from historical performance data, these methods reduce reliance on expert knowledge and can adapt to different system configurations.

Finally, we discuss methodologies that combine ML with HwPCs. These techniques represent recent advances in automated performance modeling, as they leverage low-level hardware metrics to capture detailed execution behavior. By using HwPCs as features for ML models, these approaches can identify subtle performance patterns and relationships that are difficult to capture through code analysis alone.

Our research aligns with the fourth category, as we aim to develop models that optimize TPs in parallel code-regions by leveraging HwPCs. However, existing methods often lack automation in critical aspects such as HwPC reduction, code-region identification, and TP optimization—all of which are fundamental for effectively applying ML to performance modeling. Moreover, to our knowledge, none of the existing approaches employs a systematic and principled methodology to select the most relevant HwPCs for different architectures. By addressing these gaps, our work aims to enhance the accuracy and applicability of ML-driven performance tuning.

Beyond these four categories, we note that HwPCs have proven valuable in other performance optimization contexts that do not involve ML, such as compiler-based Profile-Guided Optimization [73]. While such approaches differ from our ML-focused methodology, they demonstrate the rich performance information that HwPCs can provide, which motivates our use of HwPCs as features for ML models.

2.1 Analytical Performance Modeling

This section presents analytical approaches to performance modeling and optimization. They can be divided into two main categories: tools that employ explicit

2. Related Work

performance models to guide tuning decisions, and tools that use search algorithms to explore the parameter space. The first category relies on expert-defined models, predefined patterns, or plugin-based architectures to identify performance bottlenecks through instrumentation and runtime monitoring. The second category explores combinations of TPs using various search strategies—from evolutionary algorithms to Bayesian optimization—to efficiently navigate large parameter spaces. While analytical approaches have proven effective in many scenarios, they often require significant expertise and may struggle with the complexity of modern heterogeneous systems.

2.1.1 MATE

MATE (Monitoring, Analysis and Tuning Environment) [51, 50], is a dynamic tool for monitoring, analyzing, and tuning the performance of parallel applications at runtime. The performance analysis and tuning logic is implemented through Tunlets, which contain performance models that guide parameter modifications to improve performance. Since MATE operates dynamically during application execution, the tunable parameters are limited to those modifiable at runtime, such as the number of threads and scheduling policies in OpenMP applications [18]. The tool must maintain low overhead to enable rapid performance analysis and parameter adjustments.

MATE consists of three main components: Application Controller (AC), which executes on each host and manages application execution, instrumentation insertion, and parameter modification; Analyzer, which processes metrics collected from instrumentation and applies the Tunlet’s analysis logic to determine necessary tuning actions; and Dynamic Monitoring Library (DMLib), a shared library containing code snippets for runtime insertion into applications for monitoring and parameter modification.

Each Tunlet addresses a specific performance problem and how to improve the application performance. It comprises three elements: measure points that specify instrumentation locations, a performance model that analyzes application performance to identify problems and suggest parameter values, and tuning points that

define which parameters to modify, where to apply changes, and how to synchronize processes to maintain application consistency.

2.1.2 Periscope

The Periscope Tuning Framework (PTF) [48] was developed as part of the AutoTune European project [47], building upon Periscope [26], a performance analysis environment. PTF is an automatic performance analysis and tuning tool that addresses both performance and energy efficiency. The framework employs a plugin-based architecture where each plugin analyzes different performance aspects such as load balancing, energy consumption, data locality, and memory accesses. Multiple plugins can be applied simultaneously to explore various aspects of an application.

PTF analyzes different code-regions by executing them with multiple variants, where each variant uses different values for the TPs. After analysis, PTF generates a report containing objective metrics (such as execution time and efficiency) for each code-region and variant. Each plugin follows a defined workflow consisting of six steps: initialization of variables, tuning space, and search algorithms; scenario creation through tuning space exploration; scenario preparation by modifying tunable parameters or recompiling the application; experiment definition where scenarios are assembled and executed (multiple scenarios can run in one process for different code-regions, but scenarios for the same region require separate executions); application restart with tuning actions applied; and results processing to generate the performance report.

PTF includes several built-in plugins for tuning compiler flags, energy parameters, MPI settings, and master/worker configurations. The framework also supports user-defined plugins for custom tuning requirements.

2.1.3 Score-P

Score-P [39] is a performance measurement infrastructure jointly developed by leading HPC performance tools groups to address the fragmentation in the perfor-

2. Related Work

mance tools landscape. The infrastructure provides a unified platform for large-scale performance measurements that can be interpreted with multiple analysis tools, including HPCToolkit [1], Paraver [58], Scalasca [25], TAU [66], and Vampir [38]. By consolidating redundant basic functionality across different tools—such as source code instrumentation, profiling, and event recording for MPI—Score-P simplifies installation and reduces the learning effort for users while freeing development resources for more advanced analysis features.

Score-P consists of an instrumentation framework, several runtime libraries, and helper tools. The instrumentation framework allows users to insert measurement probes into C/C++ and Fortran codes through multiple methods: compiler instrumentation, MPI library interposition, OpenMP source code instrumentation using OPARI2, source code instrumentation via the TAU instrumentor, and user-defined macros.

The instrumentation is applied via the `scorep` command, which acts as a prefix to standard compile and link commands, automatically detecting the programming paradigm and adding appropriate compiler and linker flags.

During execution, Score-P employs a flexible memory management system that stores data in thread-local chunks of pre-allocated memory, minimizing perturbation and overhead. By default, Score-P operates in profiling mode, producing data in the CUBE4 format that provides initial insights about application hotspots and allows customization of subsequent measurement runs. Users can switch between profiling mode, tracing mode, and on-line access without recompiling or re-instrumenting the application. To manage data volume, Score-P supports filtering by region names and wildcards, as well as selective tracing that restricts recording to specific region executions. The recorded data is written to uniquely named experiment directories in formats including OTF2, CUBE4, and TAU snapshots.

Score-P addresses critical scalability limitations of previous tools, including improved handling of many MPI communicators at large scales and a tree-based reduction algorithm for identifier unification that replaces the linear complexity approach of earlier systems.

2.1.4 Scalasca

Scalasca [25] is an open-source performance analysis toolset specifically designed for large-scale parallel systems with thousands of cores. Developed by Jülich Supercomputing Centre as the successor to KOJAK [75], Scalasca supports an incremental performance analysis procedure that integrates runtime summaries with in-depth studies of concurrent behavior via event tracing. A distinctive feature is its ability to identify wait states that occur due to unevenly distributed workloads, which is crucial for scaling communication-intensive applications to large processor counts.

Scalasca supports measurement and analysis of MPI, OpenMP, and hybrid applications written in C, C++, or Fortran. The analysis workflow begins with application instrumentation, followed by execution where users can generate either summary reports (profiles) with aggregate performance metrics for function call paths, or event traces recording individual runtime events. During tracing, each process generates a local trace file. After program termination, Scalasca loads these traces into memory and analyzes them in parallel using the same number of cores as the target application. The parallel trace analysis searches for characteristic patterns indicating wait states and performance inefficiencies, classifies detected instances by category, and quantifies their significance. The resulting pattern-analysis report is similar to the summary report but enriched with higher-level communication and synchronization inefficiency metrics.

Both summary and pattern reports can be examined interactively using Scalasca’s analysis report explorer or third-party tools such as TAU’s ParaProf and Vampir. Scalasca has demonstrated scalability by successfully analyzing applications with hundreds of thousands of processes, though trace analysis is typically restricted to limited execution intervals due to memory and storage constraints.

2.1.5 Extra-P

Extra-P [14] introduces an automated approach to performance modeling that helps identify scalability bugs in parallel applications before they manifest at large scales.

The authors demonstrate that by automatically generating empirical performance models for all parts of a parallel program, they can achieve both broader coverage and faster analysis compared to manual approaches. Their approach focuses primarily on identifying scalability bugs—parts of a program exhibiting unintentionally poor scaling behavior—with emphasis on weak scaling scenarios. The method takes performance measurements (parallel profiles) from runs at different processor counts as input and produces a ranked list of program regions (kernels) based on their predicted execution time at a target scale significantly larger than those measured.

These parallel profiles can be generated using tools like Scalasca [25]. Extra-P then automatically creates empirical performance models for each program region and extrapolates them to larger scales, producing a ranked list of kernels based on their predicted execution time at target scales. This ranking effectively highlights regions with poor scaling behavior that warrant further investigation.

The workflow involves several key steps: statistical quality control to ensure reliable performance data; regression to create initial performance models for all program regions; iterative refinement to improve model quality; and extrapolation to either a specific target scale or to infinity for asymptotic behavior analysis. The approach generates both execution-time models and resource requirements models (e.g., memory usage, communication patterns), enabling developers to distinguish between computational and resource-driven bottlenecks and supporting software-hardware co-design considerations.

Using a climate simulation application as a case study, they demonstrate that scalability issues often arise in unexpected parts of the code that would typically be overlooked in traditional kernel-based analysis. The automatically generated models prove sufficiently accurate for identifying scalability bugs, sometimes matching

the quality of hand-crafted models from literature.

2.2 Autotuning with Search in the Tunable Parameters' Space

This section discusses tools that utilize search algorithms within the space defined by tunable parameter values. These tools conduct iterative searches by adjusting tunable parameters based on the chosen search algorithm, evaluating performance after each adjustment. Rather than relying on analytical models or ML, these approaches explore the configuration space using techniques such as evolutionary algorithms, simulated annealing, or multi-armed bandit strategies. The process continues until an optimal configuration is found, or a sub-optimal configuration is identified if exhaustive search is not conducted. While these methods can be effective and precise, they may require significant computational resources and time to explore large parameter spaces.

2.2.1 OpenTuner

OpenTuner [6], introduced by Ansel et al., addresses key challenges in program autotuning through a flexible, open-source tool designed for building domain-specific multi-objective program autotuners. The authors identify three major obstacles in developing effective autotuning systems: selecting appropriate configuration representations, navigating extremely large search spaces, and handling the complex landscape of configuration spaces. OpenTuner's architecture overcomes these challenges through customizable configuration representations that preserve domain-specific information, and an extensible technique framework that accommodates problem-specific search methods. The system employs sophisticated ensembles of search techniques, including the AUC Bandit meta technique, differential evolution, Nelder-Mead, and Torczon hillclimbers, to efficiently explore configuration spaces across diverse computational domains. This core innovation simultaneously deploys multiple search techniques and dynamically allocates computational resources to those performing best, using an optimal solution to the multi-armed

bandit problem. This approach enables different techniques to share results constructively through a common database, creating a more robust optimization strategy than any single technique could provide.

The authors demonstrated OpenTuner’s effectiveness by implementing autotuners for seven distinct projects across 16 benchmarks, achieving performance improvements of up to $2.8\times$ compared to previous approaches. Its pluggable architecture enables sophisticated optimization strategies while supporting various hardware platforms by abstracting configuration spaces and search mechanisms. Their work emphasizes the critical importance of domain-specific representations and search techniques in autotuning, while providing a framework that significantly reduces both manual tuning efforts and the expertise required to implement specialized systems.

2.2.2 Kernel Tuning Toolkit

Kernel Tuning Toolkit (KTT) [57], is a tool for autotuning OpenCL and CUDA applications to achieve performance portability across heterogeneous computing architectures. KTT supports two distinct tuning approaches: offline autotuning performed prior to application execution by a separate utility, and dynamic autotuning conducted during application runtime. This dual approach addresses limitations of previous frameworks that only support standalone offline tuning.

With offline tuning, KTT can search for optimal TPs for computationally demanding kernels and export these values to the build system, though this approach cannot easily adapt during execution. Dynamic tuning allows applications to execute autotuning at any time—when running on new hardware or when performance-relevant data characteristics change. Dynamic tuning operates in either blocking mode (testing multiple configurations before selecting the best one, with automatic replication of input/output arrays to prevent side effects) or non-blocking mode (where each tested kernel variant’s results are immediately used by the application).

KTT integrates directly into application code as an intermediary layer between the

2. Related Work

application and the OpenCL/CUDA API, allowing applications to decide whether to change TPs and recompile kernels (tuning mode) or simply execute optimized kernels (running mode). It collects HwPCs alongside execution time measurements, enabling detailed performance analysis and supporting advanced search strategies that can analyze profiling data to determine the relative impact of TPs on performance.

The toolkit uses preprocessor macros to enable parameterization of kernel source code, allowing TPs to control various optimizations including low-level transformations (such as loop tiling, loop unrolling, and vectorization), work-group sizes, memory layouts, and even algorithmic variants. This approach gives programmers the flexibility to define what aspects of the code can be tuned and how TPs affect the generated code.

2.2.3 Kernel Tuner

Kernel Tuner [71] is a Python-based autotuning tool designed to optimize OpenCL, CUDA, and C kernels. Developed by van Werkhoven et al., it addresses the challenge of finding optimal thread block dimensions and other optimization parameters for GPU programming. The authors emphasize that GPU kernel performance can vary dramatically based on specific combinations of thread block dimensions and optimization parameters like tiling or unrolling factors. Finding these optimal configurations typically requires searching through vast, discontinuous parameter spaces, which is time-consuming and computationally expensive.

Unlike application-specific tuners, Kernel Tuner provides a generic framework where developers can specify kernel code, tunable parameters, and execution constraints through Python scripts. The tool handles compilation, benchmarking, and automated search for optimal configurations. It implements numerous search optimization algorithms, including basin hopping, differential evolution, and the firefly algorithm. These efficiently navigate large parameter spaces by performing both compile-time and runtime optimizations simultaneously. The search algorithms accelerate the tuning process compared to brute force approaches. In one example, the authors demonstrate this by tuning a GEMM kernel on AMD Vega

Frontier Edition 71.2 $\times$ faster than with brute force search. The tool’s flexibility extends beyond single kernel optimization to more complex scenarios like tuning for computation and communication overlap and optimizing pipelines of kernels.

2.3 Tuning Methodologies Based on ML

This section examines approaches that leverage ML to automate performance optimization without explicitly relying on HwPCs as primary features. These methodologies represent a significant step toward reducing dependency on expert knowledge by learning patterns from execution data. The approaches use various application characteristics as features for their ML models, including code structure, execution patterns, workload properties, and system configuration parameters such as thread counts and processor frequencies. While these ML-based approaches successfully automate aspects of performance tuning, they typically do not capture the fine-grained hardware-level execution behavior that HwPCs provide.

2.3.1 READEX

READEX (Runtime Exploitation of Application Dynamism for Energy-efficient eX-ascale computing) [37, 65] was a European project aimed at developing dynamic autotuning tools to enhance both performance and energy efficiency in exascale computing, particularly for heterogeneous and embedded systems. A key objective of the project was to integrate ML techniques to dynamically adjust parameters based on the changing behavior of High Performance Computing (HPC) applications.

The READEX methodology involved an initial analysis phase using the Periscope Tuning Framework (PTF) [48] and a representative dataset. This phase was designed to automatically identify critical execution regions, characterize application dynamism, and estimate performance and energy efficiency for subsequent executions.

To determine optimal parameter configurations for a given system, multiple con-

2. Related Work

figurations were tested using different search strategies within PTF. The results, covering various performance and energy efficiency metrics, were then stored in a database. A classifier model was employed to categorize different execution regions into predefined scenarios. Additionally, a configuration selector mapped these scenarios to the most suitable parameter configurations. In cases where the application exhibited behavior that did not align with any known scenarios, a calibration mechanism was used to adjust the configurations accordingly.

In a related study [54], READEX explored the use of neural networks, support vector machines, and decision trees to predict optimal hardware parameters—such as the number of threads and core/uncore frequencies—for improved energy efficiency. The study, conducted on four sparse matrix algorithms, also aimed to predict energy consumption. Among the evaluated models, neural networks provided the most accurate results, with a separate model trained for each sparse matrix algorithm. READEX’s study showed high accuracy for core frequency and total energy but struggled with number of threads and uncore frequency. This highlights the challenge of predicting optimal thread configurations in complex systems.

2.3.2 BLISS

BLISS [62] applies ML to performance optimization in a way reminiscent of autotuning techniques, but with a crucial difference. Rather than exhaustively exploring the parameter search space, BLISS employs Bayesian Optimization to dynamically build performance models during runtime, eliminating the need for offline training phases.

While traditional autotuning approaches modify multiple application parameters to enhance performance, BLISS takes a more sophisticated approach. Instead of directly searching for optimal configurations, it constructs a probabilistic model that correlates input parameters with performance metrics (such as execution time). This model then guides the system toward configurations with higher probability of performance improvement.

Bayesian Optimization excels at minimizing objective functions by creating prob-

abilistic models that predict function behavior. By leveraging both input parameters and observed outputs, it enables efficient exploration of complex search spaces. This approach proves particularly valuable when evaluating all possible configurations would be computationally prohibitive [24].

The BLISS framework uses this probabilistic model to recommend parameter configurations that optimize the objective function, thereby enhancing application performance. When compared to conventional autotuning methods that rely on exhaustive searches, BLISS achieves comparable parameter configurations while requiring substantially fewer samples, making it a more efficient and practical solution for performance optimization in complex computing environments.

2.3.3 Predictive Modeling for Thread Optimization in OpenMP-based Parallelization Using ML

Yadav et al., in [77] addressed the challenge of optimizing OpenMP directives for parallel execution, with a focus on determining the optimal number of threads. The authors present an ML approach to automate this manual and time-consuming parameter tuning process.

The researchers constructed a comprehensive dataset using a Python-based loop generator program that created a variety of parallelizable loops with different array-based operations, including copying elements of 1D arrays, multiplying array elements by variables, and performing element-wise operations between arrays. The complexity of these loops was intentionally varied to simulate different computational workloads. While focusing primarily on simple computational tasks, this approach allowed for clear analysis of commonly used parallel codes.

Each loop configuration was executed five times with different numbers of threads (2, 4, 8, etc.) to ensure reliable performance measurements. The dataset captured key features affecting execution performance, including the number of threads, statements, operations, variables, arrays, and loop trip count. Categorical features were transformed using one-hot encoding to enable effective training of the regression model.

2. Related Work

The authors employed Random Forest Regression, an ensemble learning algorithm that constructs multiple decision trees and averages their predictions, making it robust to noise and high-dimensional data. Their model outperformed other regression-based methods across multiple evaluation metrics, demonstrating strong predictive accuracy. These results demonstrated that ML models could effectively optimize the number of threads in OpenMP applications.

2.3.4 Artemis

Artemis [76] is an ML-based, online performance autotuning framework designed for High Performance Computing (HPC) applications. It operates in situ by monitoring and adjusting the execution policies of parallel code-regions during runtime. Artemis enables performance portability across diverse systems by dynamically selecting the optimal execution policies based on real-time application features and performance data, with minimal developer involvement. The framework integrates seamlessly into HPC applications that use performance portability libraries such as RAJA [9] and Kokkos [21, 69]. These libraries provide abstraction layers that allow code to be written once and executed across different architectures, offering pre-packaged execution policies such as OpenMP vs. sequential execution in RAJA, or various CUDA kernel configurations in Kokkos. Artemis provides automatic tuning capabilities for these policies without requiring significant code modifications.

The core of Artemis lies in its ability to instrument parallel regions, collect performance data using HwPCs and execution time measurements, and train Random Forest-based ML models to make policy decisions for each region. During application execution, Artemis observes the parallel regions, selects different execution policy variants, and tunes these regions based on immediate performance feedback. Artemis embeds its API directly into these frameworks' implementations, allowing developers to leverage these pre-packaged execution policies or customize their own policy variants.

The framework's adaptability allows it to operate across various HPC applications, supporting both shared-memory parallelism through OpenMP, GPU acceleration

via CUDA, and distributed-memory parallelism using MPI.

Artemis optimizes performance by continuously refining execution models during application runtime. The system’s design ensures that it can operate with minimal overhead, achieving up to 47% [76] speedup in tested applications, including Cleverleaf [45, 42], LULESH [36], and Kokkos Kernels SpMV, with less than 9% monitoring overhead. It has the ability to select the optimal policy with over 85% accuracy, thus enhancing execution efficiency and achieving performance portability in HPC environments.

2.3.5 Learning to Select Scheduling Algorithms in OpenMP

In [41, 53] the authors propose a reinforcement learning (RL)-based approach for automated online selection of scheduling algorithms in OpenMP applications, extending the LB4OMP [40] and Auto4OMP [49] frameworks to address the algorithm selection problem for parallel loop scheduling.

The study compares expert-based selection methods (RandomSel, ExhaustiveSel, and ExpertSel) [49] that use predefined rules, against model-free RL-based approaches (Q-Learn and SARSA) that learn from execution feedback. The RL agents were implemented via a reusable framework (`kmp_agent_provider.h`) integrated with LB4OMP to select from a portfolio of 12 scheduling algorithms. The agents operate at the loop level with reward functions based on loop execution time and load imbalance metrics, following an "explore-first policy" that initially tests all algorithms before refining selections through learned policies.

The evaluation across six applications on three systems (720 configurations, 3,600 total executions) showed that expert-based methods provide low-overhead and stable performance but risk suboptimal selections, while RL-based methods offer greater adaptability at the cost of exploration overhead. Combining expert knowledge with RL substantially reduced learning costs and improved selection accuracy, demonstrating the effectiveness of hybrid approaches for adaptive scheduling in HPC systems.

2.3.6 Automated Algorithm Design for Autotuning Optimizers

This work [74] investigates using large language models (LLMs) to automatically generate optimization algorithms tailored to autotuning problems. The approach addresses the challenge that no single optimization method performs best across all tuning tasks, while manual algorithm design requires significant expertise. The authors introduce a framework based on LLaMEA [70], which combines LLMs with an evolutionary algorithm to generate metaheuristics. The system operates as follows: (1) an LLM generates a population of candidate optimization algorithms based on problem descriptions and search-space characteristics, (2) each candidate is evaluated within the Kernel Tuner framework using a performance score on a training set, (3) high-performing algorithms are selected for reproduction while poorly performing ones are discarded, and (4) LLM-driven mutation operators generate new candidates. The LLM proposes algorithms while the evolutionary selection mechanism ensures only high-performing candidates survive, with evaluation based entirely on measured performance in Kernel Tuner. The evaluation across four real-world autotuning applications on six hardware platforms compared LLM-generated optimizers against state-of-the-art algorithms from two contemporary autotuning frameworks. Providing application-specific and search space-specific information during generation resulted in average performance improvements of 30.7% and 14.6%, respectively. The best-performing generated optimization algorithms achieved an average 72.4% improvement over state-of-the-art optimizers, demonstrating that LLM-generated optimizers can rival and often outperform existing human-designed algorithms. The best-performing algorithms have been integrated into the Kernel Tuner framework.

2.4 Tuning Methodologies Based on ML and Hardware Performance Counters

This section focuses on approaches that combine ML with HwPCs to achieve fine-grained performance modeling and optimization. These methodologies explicitly

leverage low-level hardware metrics—such as cache misses, instruction counts, cycles per instruction, and TLB statistics—as features for their ML models, enabling them to capture detailed execution behavior that is invisible at higher abstraction levels. By combining structural code representations with HwPCs, these approaches can identify the complex interactions between code, compiler optimizations, and hardware architecture. However, existing methods in this category often lack systematic approaches for HwPC selection and reduction, limiting their effectiveness and generalizability across different applications and architectures.

2.4.1 Hardware Performance Counter Reliability

The reliability of HwPC measurements presents significant challenges for optimization methodologies that depend on accurate performance characterization. Weaver and McKee [72] conducted a comprehensive investigation into the trustworthiness of hardware HwPCs, specifically examining whether they produce consistent and accurate measurements across different experimental conditions and processor implementations.

Their study analyzed retired instruction counts—a fundamental metric used in CPI and IPC calculations—across nine different x86 processor implementations using the complete SPEC CPU 2000 and 2006 benchmark suites. Retired instruction counts are particularly suitable for cross-machine validation because they exclude speculation and cache effects, making them architecturally deterministic for a given statically linked binary. The analysis revealed that under standard experimental conditions, HwPCs can exhibit coefficients of variation up to 1.07%, with certain benchmarks showing billion-instruction discrepancies that, while appearing small in percentage terms, can significantly impact phase-behavior analysis and tool validation.

Through systematic investigation, the authors identified several sources of measurement variation, including floating-point control word instructions on Pentium 4 processors, dynamic linking effects, address space layout randomization (ASLR), and system state differences. Their work demonstrated that seemingly minor experimental factors—such as binary linking methods, memory allocation patterns,

and benchmark preparation procedures—can substantially affect counter accuracy.

However, their findings also showed that with rigorous measurement protocols and careful control of experimental conditions, observed errors can be reduced to less than 0.002% for all benchmarks across both SPEC suites. This dramatic improvement in measurement reliability was achieved through simple methodology changes, including using statically linked binaries, disabling ASLR, controlling system state, and ensuring consistent benchmark compilation. These results established that x86 retired instruction HwPCs, when used correctly, can provide highly accurate representations of hardware behavior, making them suitable for architectural research and performance analysis. The consistency of results across multiple processor generations further validated that HwPC-based measurements remain comparable across different hardware platforms, enabling longitudinal studies and cross-generational performance comparisons.

2.4.2 Hardware Performance Counters for GPU Autotuning

While autotuning typically achieves significant performance improvements, the search process can introduce unacceptable overhead when tuning spaces are vast with predominantly poorly-performing configurations, or when autotuning must be repeated frequently due to changes in input data or hardware migration. This challenge is particularly acute for GPU accelerators, where performance portability across diverse architectures and input characteristics requires continuous adaptation.

State-of-the-art autotuning methods typically learn direct relationships between TPs and execution performance. However, these learned relationships become invalid when the hardware platform or input data changes, requiring the autotuning process to restart from scratch. In contrast, Filipovič et al. [23] exploit a key insight: while the TP-to-performance relationship varies significantly across different platforms and inputs, the relationship between TPs and HwPCs remains relatively stable across these variations.

The method distinguishes between two types of HwPCs: $HwPC_{Stress}$, which mea-

2. Related Work

sure stress on processor subsystems, and $HwPC_{Ops}$, which measures the amount of operations on processor subsystems. Their key insight is that while the relationship between TPs and execution time varies significantly across different hardware and inputs, the relationship between TPs and normalized $HwPC_{Ops}$ values remains relatively stable.

The proposed profile-based searcher mimics iterative optimization performed by developers, who use profilers to identify bottlenecks and modify code accordingly. The method operates in two phases: a training phase where ML models capture how TPs influence HwPCs, and an autotuning phase where the method iteratively profiles code, analyzes bottlenecks using an expert system, and uses the trained model to determine which tuning configurations will change HwPCs in the desired direction. Critically, models built using a particular GPU and input can be applied to speed up autotuning on different GPUs or with different inputs, enabling cross-platform and cross-input performance portability.

The framework was implemented in the Kernel Tuning Toolkit (KTT) and evaluated using five benchmark kernels across four NVIDIA GPU architectures spanning multiple generations. The experimental results demonstrated that the profile-based searcher consistently outperforms random search when using models created on different GPUs, different inputs, or smaller tuning spaces. The method also proved superior to state-of-the-art approaches, including the optimization-based Basin Hopping searcher in Kernel Tuner and the regression-tree model in Starchart, converging faster to near-optimal configurations in most cases. The evaluation confirmed that HwPCs can significantly reduce the number of empirical tests required during autotuning, with the real-time convergence typically superior to existing methods.

2.4.3 APARF

APARF (Automatic, Portable, and Adaptive Runtime Feedback-Driven Framework) [61] is a system that integrates low-level tasking runtime APIs, profiling, and ML to optimize task scheduling in parallel applications. Unlike static heuristics commonly used in traditional scheduling approaches, APARF dynamically

2. Related Work

adapts to changing execution conditions through runtime feedback. This adaptability is particularly valuable in heterogeneous computing environments, where optimal scheduling decisions depend on workload characteristics and hardware configurations.

The framework is trained using multiple task-based applications from the Barcelona OpenMP Tasks Suite (BOTS) [19], which exhibit diverse behaviors and task characteristics, ensuring a well-balanced and representative dataset. To capture a wide range of execution scenarios, each application is run under varying problem sizes, thread counts, compiler flags, and task scheduling configurations.

APARF focuses on two key factors for task scheduling: load balancing and data locality. Load balancing is assessed by measuring execution times across application threads, while data locality is characterized using HwPCs, including cache accesses and misses, cycles-per-instruction ratios, and TLB misses. The ML component of APARF employs an artificial Neural Network (ANN) to classify applications into different categories, each corresponding to a distinct task pool configuration.

The effectiveness of APARF was evaluated by comparing its scheduling decisions against the default task scheduling strategies of various compilers. Out of 120 previously unseen test cases, only 8 were misclassified, primarily involving small problem sizes with erratic behavior. The framework achieved an overall accuracy of 93%, resulting in a 25% performance improvement compared to the default compiler configurations.

2.4.4 Piecewise Holistic Autotuning with CERE

The CERE project, introduced in [60], aims to automatically optimize OpenMP applications by decomposing them into smaller code-regions called *codelets*, which are mapped to OpenMP parallel regions. These codelets are then grouped using clustering techniques to identify those with similar performance profiles. CERE combines both static metrics, such as code characteristics, and dynamic metrics, including HwPCs and memory bandwidth, to generate performance signatures for each codelet.

2. Related Work

The approach involves applying performance tuning to a representative codelet within each cluster, and subsequently, applying the same tuning strategy to all codelets in that cluster. Once tuning is applied, the codelets are extracted into separate code files, compiled individually with the appropriate TPs, and then integrated into a hybrid binary that contains both the original application and the optimized codelets.

The results of CERE demonstrate significant improvements, with a $6.55\times$ reduction in the time required to evaluate the codelets compared to evaluating the entire application. Additionally, CERE achieved an impressive accuracy of around 93.66% when applied to the NAS benchmark, highlighting its effectiveness in optimizing performance for parallel OpenMP applications.

This approach offers a practical method for improving parallel application performance without the need for full application-level tuning, making it highly efficient for large-scale systems.

2.4.5 Pattern-based Autotuning of OpenMP Loops Using Graph Neural Networks

Dutta et al. in [20] develop a pattern-guided deep learning approach for optimizing OpenMP code. Their methodology employs Graph Neural Networks (GNNs) to identify common parallel loop patterns and determine optimal runtime parameters. By modeling code as flow multi-graphs and capturing execution behavior through HwPCs, they developed a system that identifies parallel loop patterns with 91% accuracy.

The researchers focused specifically on loop-level parallelism patterns, including *do-all*, *reduction*, and *stencil* patterns, which are prevalent in scientific applications. Their approach demonstrates significant performance improvements, achieving geometric mean speedups of $1.1\times$ on single-socket systems and $4.7\times$ on multi-socket NUMA systems compared to default OpenMP configurations, approaching the theoretical maximum speedups of $1.27\times$ and $4.93\times$ respectively from brute-force optimization.

The authors use Relational Graph Convolutional Networks (RGCNs) to model code graphs, combining these structural embeddings with HwPCs to guide parameter tuning. This approach differs from traditional methods that rely on expert-designed heuristics and rules for pattern detection. While the approach incurs training overhead, it reduces the number of executions needed during inference (only two) compared to brute-force approaches that would require 168 and 64 executions for comparable tuning experiments. The authors position their work as complementary to existing tools like DiscoPoP [43], which can identify certain patterns (*do-all* and *reduction*) but fails to detect others (*Stencil*).

2.5 Comparison of the presented approaches

The previously described analytical performance models are effective for identifying performance bottlenecks and analyzing scalability. However, these models remain fundamentally limited by their dependence on expert-defined models and predefined performance patterns. Tools such as MATE, Periscope, and Scalasca require significant domain expertise to interpret results and make tuning decisions. While Extra-P advances the field by automating performance model generation, it still relies on expert knowledge for model specification and interpretation. Moreover, these analytical approaches typically focus on performance analysis and identification of bottlenecks rather than automated parameter optimization. These models could be adapted to address more complex scenarios, but analytical models require extensive knowledge and time to be generated properly, particularly as systems become increasingly heterogeneous.

In the case of autotuning tools that employ search algorithms to explore the tunable parameter space, such as OpenTuner, KTT, and Kernel Tuner, there is significant potential for performance optimization. However, the overhead introduced into the application to find the proper parameter configuration should be taken into account. Even with sophisticated search strategies like multi-armed bandits or Bayesian optimization, these approaches require multiple application executions to converge on optimal configurations. Trying different configurations may deteriorate performance during the search process, and this overhead can be coun-

2. Related Work

terproductive, even if at some point the proper configuration is found. This is especially true for applications with short execution times or when frequent re-tuning is necessary due to changing workload characteristics.

Therefore, due to the complexity of generating new analytical models and the overhead of exhaustive search approaches, new approaches are necessary. Multiple ML based approaches for performance tuning have been presented, including READEX, BLISS, Artemis, APARF, CERE, and the GNN-based approach by Dutta et al. These approaches show that using ML is an ongoing and important topic for the automatic generation of performance models without the intervention of performance experts. ML-based approaches without HwPCs, such as READEX, BLISS, and Artemis, demonstrate that models can effectively predict optimal configurations based on application features such as workload characteristics, input parameters, and system configuration. However, these approaches operate at a relatively high level of abstraction, using features like number of threads, problem size, and execution time without capturing the fine-grained hardware-level execution behavior that determines actual performance.

The approaches that combine ML with HwPCs—including the GPU autotuning method by Filipovič et al., APARF, CERE, and the work of Dutta et al.—recognize the value of low-level hardware metrics for performance modeling. HwPCs provide direct visibility into architectural events such as cache behavior, instruction execution patterns, memory access efficiency, and branch prediction accuracy. These metrics capture the complex interactions between code, compiler optimizations, and hardware architecture that are invisible at higher abstraction levels. The work by Weaver and McKee on HwPC reliability demonstrates that with rigorous measurement protocols, HwPCs can provide highly accurate representations of hardware behavior, with observed errors reduced to less than 0.002% for retired instruction counts. This level of reliability is crucial for the effectiveness of HwPC-based optimization approaches. Furthermore, their findings demonstrate that HwPC measurements remain consistent across multiple processor generations, validating the comparability of results across different hardware platforms. However, their work also highlights that subtle changes in experimental setup—such

2. Related Work

as binary linking methods, memory allocation patterns, and system state—can significantly impact measurement accuracy if not properly controlled. This sensitivity to experimental conditions underscores the importance of establishing robust measurement methodologies for HwPC-based optimization approaches.

In performance modeling, the selection of metrics which correctly describe the behaviour of an application is an important issue to take into consideration. In the case of analytical models generated by experts, this selection is limited to the metrics which experts consider the most representative according to their knowledge. This limited selection in analytical models is important in order to reduce the time required when researching relationships between metrics and performance. In ML more metrics can be used as the relationships are automatically inferred, allowing the model to infer more knowledge about the behaviour of an application. However, the presented works seem to decide the metrics used for training in the same way as analytical models, using a limited number of HwPCs the authors consider good for the model to generate, instead of exhaustively evaluating which of the available HwPCs should be used. For instance, Filipovič et al. manually categorize GPU counters into stress (measuring subsystem pressure) and operations (measuring operation volume) types, and APARF selects cache-related metrics and TLB misses for task scheduling.

These choices, while effective for their specific use cases, are based on expert intuition about which HwPCs are most relevant rather than systematic evaluation across the full range of available HwPCs. This arbitrary selection of HwPCs introduces several problems. Different HwPCs may be relevant for different types of performance bottlenecks, architectural configurations, or code patterns. Without systematic evaluation, researchers cannot determine whether their chosen HwPCs adequately capture the performance characteristics of interest or whether crucial information is being overlooked. Furthermore, the selected HwPCs may not generalize well across different applications, architectures, or workload characteristics, limiting the portability and applicability of the resulting models.

As stated previously, the large number of available HwPCs per architecture, makes manual analysis impractical, necessitating ML-based approaches to automatically

2. Related Work

identify relevant performance patterns from this high-dimensional data. However, to effectively leverage ML for performance modeling, these approaches require balanced and representative datasets containing HwPC information from diverse execution scenarios. Without such datasets, ML models risk being biased toward overrepresented configurations and failing to capture the full spectrum of performance behaviors across different code-regions and TP settings.

Despite this fundamental requirement, we were unable to find adequate consideration of dataset balance and representativeness in most ML-based performance tuning approaches. CERE addresses this partially through its approach of decomposing applications into smaller code-regions called codelets, which are then grouped using clustering techniques based on performance signatures generated from both static code characteristics and dynamic metrics including HwPCs. By identifying codelets with similar performance profiles, CERE can apply tuning to a representative codelet within each cluster and propagate the same strategy to all members of that cluster. This clustering-based approach ensures some degree of representativeness by selecting samples that capture different performance behaviors, though it operates within a single application rather than across diverse workloads.

APARF takes a broader approach to dataset construction by explicitly training across multiple task-based applications from the Barcelona OpenMP Tasks Suite (BOTS) [19], which exhibit diverse behaviors and task characteristics. To capture a wide range of execution scenarios, each application is executed under varying problem sizes, thread counts, compiler flags, and task scheduling configurations. This deliberate construction of training data across multiple applications with diverse characteristics ensures a well-balanced and representative dataset that captures different execution patterns and performance behaviors. However, most other ML-based performance tuning works do not explicitly discuss dataset balance and representativeness. The effectiveness of ML models critically depends on how training data is collected across different TP configurations. When optimizing TPs like number of threads, naive sampling strategies often lead to overrepresentation of certain configurations—particularly the maximum number of available

2. Related Work

threads—while undersampling potentially optimal intermediate values. Models trained on such skewed data tend to favor the overrepresented configurations in their predictions, even when alternative configurations would deliver better performance for specific workloads.

Other fundamental limitation that existing approaches fail to address adequately is portability across different hardware architectures. Performance characteristics and optimal configurations are inherently architecture-dependent—what works well on one processor may perform poorly on another due to differences in cache hierarchies, instruction pipelines, memory systems, and execution resources. Moreover, the available HwPCs vary significantly across architectures: each processor family exposes different performance monitoring capabilities through its performance monitoring unit. While Filipovič et al. demonstrate that their GPU autotuning approach can transfer models across different NVIDIA GPU architectures by exploiting the stable relationship between TPs and normalized HwPCs, this cross-architecture capability remains limited to GPUs from the same vendor. Current approaches typically target a single architecture family or require manual reconfiguration and retraining when moving to fundamentally different hardware. For instance, a model trained with specific HwPCs on Intel processors cannot directly transfer to AMD or ARM processors, or from CPUs to GPUs, as these expose different sets of HwPCs. This lack of portability severely limits the practical applicability of existing ML-based performance tuning approaches in heterogeneous computing environments, which are increasingly common in modern HPC systems.

The reviewed ML-based performance tuning approaches exhibit three limitations that restrict their effectiveness and applicability. First, existing works rely on manual, expert-driven selection of HwPCs rather than employing methodologies to identify the most relevant HwPCs. Second, current methodologies lack portability across fundamentally different hardware architectures, requiring manual reconfiguration when moving between different processor families or vendors. Third, most approaches fail to adequately address the construction of balanced and representative training datasets, leading to models biased toward overrepresented configu-

2. Related Work

rations.

Unlike the aforementioned works, this thesis addresses identified limitations through two principal contributions.

First, it introduces an ML-based methodology for automatic HwPC selection and reduction that identifies a minimal yet highly informative set of HwPCs for performance modeling. The methodology employs principled feature selection techniques specifically adapted to the characteristics of HwPC data and performance modeling requirements. It also incorporates rigorous measurement protocols to ensure consistent and accurate HwPC measurements across different experimental conditions. This methodology substantially reduces monitoring overhead while maintaining or improving prediction accuracy, and it automates the determination of optimal TP configurations without requiring a manual effort for performance analysis and tuning.

Second, this research develops a portable optimization methodology that demonstrates effectiveness across diverse hardware architectures, including both CPUs and GPUs from different vendors. Rather than training architecture-specific models with fixed HwPC sets or limiting portability to within a single vendor’s product line, our approach dynamically selects appropriate HwPCs based on the target architecture’s available performance monitoring capabilities. This design maintains consistent prediction accuracy across different computing platforms without requiring manual architecture-specific modifications, enabling seamless optimization in heterogeneous environments where existing approaches would necessitate significant reworking for each new platform.

In our work we tackle the problems of how to build balanced and representative datasets for parallel code-regions, how to automatically select relevant HwPCs for monitoring and tuning, rather than relying on expert intuition, and how to ensure portability across fundamentally different hardware architectures. The methodology’s portability extends beyond hardware to different parallel programming models and application domains, providing a truly general-purpose solution to performance tuning.

2. Related Work

Finally, unlike existing approaches that require manual intervention at various stages of the optimization process, our methodology achieves full automation across the entire workflow—from HwPC reduction and code-region identification to TP optimization—enabling performance tuning without manual intervention at any stage.

Chapter 3

Background

This chapter provides the context for the presented research and outlines the various techniques employed in this thesis. It begins with a description of the previous work and how the HwPC datasets were constructed. Following this, the chapter offers a comprehensive overview of the ML algorithms and methodologies utilized throughout this research. Finally, it presents the ensemble methodology that combines multiple models to achieve robust predictions, along with the evaluation metrics used to assess the model performance.

3.1 Previous Work

This thesis builds upon the foundation established by Alcaraz et al. through two complementary studies that laid the groundwork. The first study, [4], proposed a methodology based on Principal Component Analysis (PCA) and correlation analysis to reduce the number of Hardware Performance Counters (HwPCs) for characterizing OpenMP parallel regions. Instead of relying on comprehensive but impractical HwPC measurements, this paper introduced a methodology to identify the most informative subset of HwPCs. Through extensive experimentation with OpenMP templates and 58 preset HwPCs, the paper successfully demonstrated that HwPC multiplexing could be used to measure a reduced set of performance indicators with high accuracy, thereby opening new possibilities for runtime per-

3. Background

formance characterization and optimization. The methodology was successfully evaluated using the STREAM [46] benchmark for various configurations.

This research was followed by the second study, [5], which developed a complementary approach, focusing on the challenge of optimizing thread allocation in parallel computing environments through ML methodologies. A key innovation of this study was its approach to handling dataset imbalances, which is a common challenge in ML-based performance optimization. Imbalances appeared when the dataset had an uneven class representation, with some underrepresented classes. This skewed ML methodologies to predict the majority class, leading to poor performance in the minority class, which is crucial for tasks such as fraud detection, medical diagnosis, and performance tuning. This study introduced a holistic methodology for constructing balanced datasets of OpenMP code-region patterns, addressing the inherent complexities of performance parameter tuning. By using HwPCs to characterize code-region executions and applying correlation analysis, this study developed a methodology to determine whether an OpenMP parallel region pattern covered previously unrepresented portions of the feature space that were not represented by the training set. Then, by combining Random Forest algorithms with binary classification methodologies, the authors proposed a model capable of predicting the optimal number of threads for different parallel regions. The methodology was validated using code-regions from the STREAM [46] and PolyBench [79, 78] benchmarks, demonstrating its potential for generating balanced datasets and creating predictive models for thread number optimization.

Despite the innovative methodologies proposed by Alcaraz, there were still limitations that constrained their broader applicability. The approach to HwPC reduction required substantial manual intervention and was effective only when the number of HwPCs and code-regions were small. This made it inherently non-scalable when confronted with a large number of HwPCs and diverse code-regions. This manual selection process introduced significant overhead and reduced the potential for automated adaptive performance optimization.

The number of threads prediction approach revealed additional methodological constraints, particularly evident in its performance across different code-region

3. Background

types. Notably, the **Stride** code-regions consistently exposed weaknesses in the prediction model, systematically underestimating the optimal number of threads. This limitation suggested that the proposed ML methodology may have lacked sufficient generalizability, indicating potential gaps in the feature representation or the model’s ability to capture the nuanced performance characteristics of certain computational patterns. Although the **Stride** code-regions demonstrated the most pronounced deficiencies, similar weaknesses were observed across the entire code-region spectrum, albeit to varying degrees.

A critical analysis of these prediction failures revealed a methodological flaw: the HwPCs utilized for code-region identification were simultaneously employed for TP tuning. To directly address this limitation, a bifurcated HwPC selection strategy was proposed that employed distinct sets of HwPCs for the separate tasks of code-region identification and TP prediction, thereby enhancing the model’s discriminative capability and predictive accuracy.

The studies by Alcaraz et al. established that HwPCs can effectively characterize parallel code-regions and that ML models can predict optimal thread configurations. However, they also revealed two key limitations that needed to be addressed: the HwPC reduction process required manual intervention and lacked scalability, and using the same HwPCs for both code-region identification and TP optimization introduced bias that hurt prediction accuracy. These findings motivate the need for a fully automated, scalable methodology that employs distinct HwPC subsets for each task, laying the groundwork for the methodology developed in this thesis.

3.2 Dataset Construction

This section focuses primarily on collecting the HwPCs that serve as features to identify OpenMP code-regions. The methodology must be applicable across various architectures, as the ultimate goal is to produce ML models that are hardware-architecture independent. To ensure architecture independence, a representative and balanced dataset must be constructed for each architecture, capturing patterns

3. Background

commonly found in parallel applications.

To this end, we adopt the approach introduced in [3, 4] for systematically collecting Hardware Performance Counters (HwPCs) for a given set of parallel regions. This method takes into account all possible combinations of architectural characteristics (e.g., number of cores, memory hierarchy), region characteristics (e.g., data layout and size), compiler optimizations, and parallelization strategies (e.g., number of threads, affinity). Consequently, characterizing each code-region requires a significant number of executions.

3.2.1 Preset Events and Compatibilities

Alcaraz et al. [4] developed a system for obtaining HwPCs using the Performance Application Programming Interface (PAPI) [52, 44, 35, 13]. This tool provides a consistent interface and methodology for accessing HwPCs found in most major microprocessors, enabling near real-time observation of the relationship between software performance and processor events.

When working with diverse architectures, the measurable HwPCs available for each system often differ. Preset events are standardized HwPCs that are typically available across multiple processor platforms, whereas native events are architecture-specific HwPCs that may not be available on other systems. To generalize the methodology, this research employs preset events for each system rather than native events, ensuring cross-architecture generalizability. While this approach may reduce the total number of available events, it significantly enhances generalization capabilities across architectures.

After compiling the list of events for each architecture, compatible groups must be determined to optimize measuring precision. Modern processors have a limited number of registers for simultaneously reading HwPCs [3]. Furthermore, incompatibilities exist when accessing certain HwPCs concurrently.

To overcome the limitation on the number of HwPCs that can be monitored concurrently, the execution of each region configuration is repeated for each group of compatible events; the resulting measures are then concatenated to create a sin-

gle, comprehensive characterization of the region. A reduced number of events per group allows for increased measuring time per group, thereby enhancing precision [3].

3.2.2 Problem Sizes

After establishing the HwPC groups, appropriate problem sizes must be determined to effectively stress the different memory hierarchies available in each architecture. *Stressing* a memory level means selecting problem sizes that force the application to utilize that specific level of the memory hierarchy—from fast on-chip caches (L1, L2, L3) to slower off-chip main memory (RAM)—thereby revealing how code-regions perform under different memory stress scenarios. This is essential because the performance characteristics captured by HwPCs vary significantly depending on whether data fits in cache or requires accessing main memory, and optimal TPs often depend on these memory access patterns.

In [5], Alcaraz et al. presented a methodology that required problem sizes to be directly proportional to memory capacity at each level of the memory hierarchy. Problem sizes are categorized as follows:

1. **Inside Processor Memory:** For memory levels residing within the processor (L1, L2, L3 caches), problem sizes are proportional to the number of physical cores in one processor:
 - (a) **Private Cache Levels:** Typically, each processor core contains private memory accessible by threads allocated to that core, specifically L1 and L2 caches. To stress these private cache levels, multiple problem sizes are created proportional to available resources, defined by multiplying the number of private cache levels by the number of cores in a processor.

For each cache level, problem sizes are constructed starting with the size of one private cache and subsequently multiplied by increasing core numbers until reaching the accumulated size of all private caches at that level.

3. Background

(b) **Shared Cache Levels:** Processors also feature shared memory across all cores, typically in the form of a shared L3 cache. To effectively stress shared cache levels, problem sizes must exceed the accumulated size of lower-level caches while remaining below the maximum shared memory capacity of the cache level. This approach ensures that lower cache levels cannot provide sufficient memory while preventing access to higher levels in the memory hierarchy.

2. **Outside Processor Memory:** For memory levels external to the processor, problem sizes must be proportional to the number of processors in the system rather than the number of physical cores:

(a) **Main Memory:** The largest problem sizes target main memory and are proportional to the number of processors plus one. The initial problem size exceeds the capacity of the last cache level of the processor. Subsequent problem sizes are obtained by progressively doubling memory requirements, with the final size reaching 1.5 times the total size of the last level cache across all processors.

3.3 Machine Learning

Machine Learning (ML) is a subfield of artificial intelligence that focuses on developing models capable of identifying patterns in data to make predictions. These models are constructed using algorithms, statistical techniques, and datasets to extract meaningful insights from information.

Machine learning methods can be broadly categorized based on the presence or absence of labeled data [28]:

- **Supervised Learning:** In this approach, each data sample is associated with one or more labels that serve as the target output for prediction. The algorithm learns to infer relationships between input features and labels by analyzing labeled examples.
- **Unsupervised Learning:** In contrast, unsupervised learning deals with

3. Background

data that lacks predefined labels. The model is tasked with identifying hidden structures, patterns, or groupings within the dataset without explicit guidance.

In this thesis, only supervised learning methods are employed. The following subsections provide a detailed discussion of these approaches and their common implementations.

3.3.1 Logistic Regression with Regularization

The following discussion examines logistic regression [17] and builds upon this foundation to explain L1 regularization [63, 67], L2 regularization [33], and their combination into the Elastic Net [80].

Logistic regression employs the sigmoid function to produce probability values ranging from 0 to 1. In binary classification tasks with classes labeled as 0 and 1, a threshold of 0.5 is typically applied: outcomes with probabilities greater than or equal to 0.5 are classified as class 1, while those below 0.5 are classified as class 0. The sigmoid function is formulated as:

$$h(z) = \frac{1}{1 + e^{-z}}, \quad (3.1)$$

where $h(z)$ represents the predicted probability, and z is the linear combination of input features and their corresponding coefficients:

$$z = \beta_0 + \beta_1 x_1 + \cdots + \beta_n x_n, \quad (3.2)$$

In this formulation, β_0 denotes the intercept term, $\beta_1, \dots, \beta_n$ represent the coefficients to be optimized, and $x_1, \dots, x_n$ are independent variables or features.

For multiclass classification problems, the one-versus-all (also known as one-versus-rest) strategy is implemented. This approach constructs multiple binary classifiers, with the number of classifiers corresponding to the number of classes. Each classifier focuses on distinguishing one specific class from all others, treating the selected class as 1 and all remaining classes as 0. During prediction, the class corresponding

3. Background

to the classifier that produces the highest probability is selected. Model performance is evaluated by measuring the discrepancy between predicted and actual values. The objective is to minimize this error through an appropriate cost function. For logistic regression with regularization, the cost function is formulated as:

$$L(\lambda_1, \lambda_2, \hat{\beta}) = \frac{1}{n} \sum_{i=1}^n [-(y_i \log(h_i) + (1 - y_i) \log(1 - h_i))] + \lambda_1 \sum_{j=1}^m |\hat{\beta}_j| + \lambda_2 \sum_{j=1}^m \hat{\beta}_j^2, \quad (3.3)$$

where n represents the number of training samples, y_i is the true label for the i th sample, and h_i is the corresponding predicted probability. The first term represents the negative log-likelihood, while the second and third terms constitute the Lasso (L1) and Ridge (L2) regularization penalties, respectively. Together, these regularization terms form the Elastic Net penalty.

Both regularization terms include a hyperparameter λ , which controls the strength of the penalty applied to the coefficients. When $\lambda = 0$, regularization is effectively removed, resulting in standard logistic regression. The fundamental difference between L1 and L2 regularization lies in their effect on coefficient values. Lasso regularization (L1) penalizes the absolute values of coefficients, enabling coefficient shrinkage to exactly zero, effectively performing feature selection by eliminating less important features from the model. In contrast, Ridge regularization (L2) penalizes the squared magnitude of coefficients, which reduces their size but does not shrink them exactly to zero, even with large λ values.

The Elastic Net [80] combines both L1 and L2 regularization techniques. To control the balance between these penalties, an additional parameter α is introduced:

$$\alpha = \frac{\lambda_2}{\lambda_1 + \lambda_2}, \quad (3.4)$$

This formulation leads to the following optimization problem:

$$\hat{\beta} = \underset{\hat{\beta}}{\operatorname{argmin}} \sum_{i=1}^n (y_i - \hat{\beta}^T x_i)^2, \quad \text{subject to } (1 - \alpha) \sum_{j=1}^m |\hat{\beta}_j| + \alpha \sum_{j=1}^m \hat{\beta}_j^2 \leq t \quad (3.5)$$

3. Background

where t is a constraint parameter. Notably, when $\alpha = 1$, Elastic Net is equivalent to Ridge Regression, and when $\alpha = 0$, it is equivalent to Lasso Regression. This flexibility allows Elastic Net to leverage the advantages of both regularization techniques.

3.3.2 Tree Based Models

The following discussion explores tree-based ensemble methods, specifically Random Forest [32, 11] and XGBoost [16]. These approaches represent two distinct ensemble paradigms: bagging and boosting, respectively. Tree-based ensemble models combine multiple decision trees to produce predictions with greater accuracy than any single decision tree could achieve independently. The underlying principle posits that an aggregation of weak learners can collectively function as a strong learner. The following sections detail these two ensemble techniques.

Random Forest Classifiers

Random Forests are an ensemble method based on Bootstrap Aggregation (bagging), which operates as a parallel learning approach. The algorithm generates multiple random training samples from the original dataset through bootstrapping. These bootstrap samples serve as independent training sets for individual decision trees within the ensemble.

Each bootstrap sample is created with replacement, allowing data points to appear in multiple samples. This sampling technique aims to capture the distribution characteristics of the full dataset while introducing variation across individual training sets. The size of each bootstrap sample typically equals that of the original dataset, though approximately one-third of the original observations are excluded from each sample—these excluded observations are termed "out-of-bag" samples and serve an important role in model validation.

A key feature distinguishing Random Forest from simple bagging is the introduction of additional randomness during tree construction. At each node split, the algorithm considers only a random subset of features rather than the entire feature set. This feature subsampling, typically using $\sqrt{p}$ features (where p rep-

3. Background

resents the total number of features), reduces correlation between individual trees and enhances the ensemble's generalization capabilities.

The training process occurs in parallel, with each decision tree independently constructed on its respective bootstrap sample. The parallel nature of this algorithm makes it highly amenable to distributed computing environments, offering significant computational advantages for large-scale applications.

The final prediction is derived through an aggregation step that combines individual tree outputs. For regression problems, this aggregation typically involves averaging the predictions across all trees. For classification tasks, majority voting determines the final class assignment. This democratic approach to prediction helps mitigate overfitting and reduces variance, yielding more robust predictions compared to individual decision trees.

Random Forests offers several advantages beyond improved prediction accuracy. The algorithm provides built-in estimates of feature importance, facilitates partial dependence analysis, and generates proximity measures between observations. Additionally, the out-of-bag samples enable internal cross-validation without requiring separate validation datasets.

Figure 3.1 illustrates the Random Forest bagging process. The training data generates multiple bootstrap samples, each training an independent decision tree in parallel. Random feature subsampling at each node ensures diverse decision boundaries across trees. For prediction, a new instance is processed by all trees simultaneously. Each tree produces a class prediction, and the final classification is determined through majority voting. This aggregation reduces variance by averaging errors across decorrelated trees, yielding more robust predictions than individual trees. The parallel architecture makes Random Forest highly scalable for high-dimensional datasets such as comprehensive HwPC measurements.

XGBoost

XGBoost (eXtreme Gradient Boosting) represents an optimized implementation of Gradient Boosted Trees rather than a novel model formulation. To comprehend

3. Background

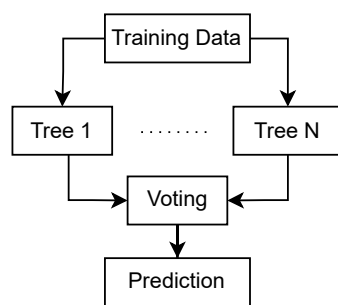


Figure 3.1: Bagging process.

XGBoost thoroughly, it is necessary to examine the progression from basic boosting to gradient boosting and subsequently to gradient boosted trees.

Boosting constitutes a sequential ensemble method that iteratively combines weak models to form a strong predictive model. The process begins by creating a random subset from the complete dataset, with all data points initially assigned equal weights. A base model is trained on this sample and generates predictions for the entire dataset. The algorithm then computes prediction errors and increases the weights of incorrectly predicted instances for subsequent model iterations. Each new model thus focuses more intently on rectifying the errors of its predecessors.

Figure 3.2 illustrates the boosting process and prediction workflow. Unlike Random Forest’s parallel approach, boosting constructs trees sequentially. The training data is used to train the first tree, which generates predictions and computes errors. Each subsequent tree is trained on a modified dataset where incorrectly predicted instances receive increased weights, forcing the new tree to focus on correcting previous mistakes. This sequential process continues through all trees, with each learning from the residual errors of its predecessors. For prediction, a new instance is processed through all trees sequentially. Each tree contributes its prediction, and these are aggregated through weighted voting to produce the final classification. This sequential error-correction mechanism enables the ensemble to progressively refine predictions, with later trees targeting patterns that earlier trees missed, resulting in a robust learner that systematically addresses collective weaknesses.

3. Background

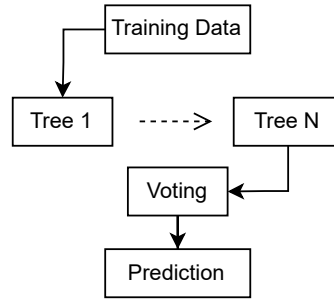


Figure 3.2: Boosting process.

The generalization process optimizes an arbitrary differentiable loss function. For classification tasks, loss functions such as log loss (binary cross-entropy), multi-class cross-entropy, or exponential loss are employed. In binary classification, logistic loss is particularly prevalent, as it directly models the probability of class membership. For multi-class problems, softmax cross-entropy serves as the standard loss function, enabling the model to output probability distributions across all possible classes.

The objective is to derive predictions that minimize the selected loss function. This minimization employs the Gradient Descent algorithm, which iteratively updates model predictions according to a specified learning rate. The process continues until the sum of residuals approaches a minimum (ideally zero), and the model's predicted values closely approximate the ground truth values. Upon completion of each model's training phase, a subsequent model is trained on the gradient of the error with respect to the loss predictions of the previous model. This approach aims to correct the errors of the preceding model. The iterative process fits each new model to the gradient of the error of the updated sum. Applying the gradient boosting methodology to decision trees yields Gradient Boosted Trees. This implementation involves two critical parameters: the number of leaves in each tree and the weights assigned to each leaf.

A significant challenge in decision tree construction concerns node splitting. Conventional implementations employ a "greedy" approach that evaluates potential splits across all features. This requires computing the resultant loss for each candidate split and selecting the tree structure that minimizes the loss function most

3. Background

effectively. After determining the optimal tree structure, weights must be assigned to each node to minimize the loss function. These weights typically correspond to the mean of the ground truth values associated with observations at each node.

XGBoost addresses computational inefficiencies inherent in standard Gradient Boosted Trees implementations, particularly when dealing with high-dimensional feature spaces where the number of potential splits becomes prohibitively large. For each candidate split, computing the associated loss function represents a significant computational burden that can render traditional implementations impractical for large-scale applications.

XGBoost mitigates this computational complexity through what its authors term an *approximate algorithm*, contrasting with the exact greedy algorithm of earlier implementations. This approach reduces the search space of potential feature splits by initially proposing candidate splitting points according to percentiles of feature distribution. The algorithm subsequently maps continuous features into discrete buckets defined by these candidate points, aggregates the relevant statistics, and identifies the optimal solution from among the proposals based on the aggregated statistics [16]. Additionally, XGBoost incorporates several technical innovations that enhance both computational efficiency and predictive performance, including a regularization term to prevent overfitting, a sparsity-aware split finding algorithm, and a distributed computing mechanism that facilitates parallel processing across multiple machines.

3.3.3 TabNet

TabNet [7] represents a specialized Deep Neural Network architecture designed to emulate Decision Tree-based learning paradigms. This architecture was developed with an emphasis on interpretability and efficient feature selection mechanisms.

To achieve interpretability, TabNet employs sequential attention mechanisms that dynamically select relevant features during each decision step. The feature selection process operates on an instance-wise basis [7], allowing the model to adapt its feature utilization patterns uniquely for each input sample. Notably, TabNet

3. Background

forgoes global normalization in favor of batch normalization techniques, which stabilizes training while preserving the relative importance of features.

As observed in Figure 3.3, the architectural foundation of TabNet consists of multiple decision blocks referred to as *Steps*—a configurable hyperparameter that determines model complexity. In an approach analogous to the tree based classifiers previously discussed, each *Step* contributes an equal weight to the final classification decision, effectively simulating a voting mechanism across multiple decision pathways.

Each *Step* comprises two primary components: the Feature Transformer and the Feature Selection module. The Feature Transformer implements a multi-layered neural network architecture where certain layers are shared across *Steps* while others remain *Step*-specific. Each layer within this component follows a consistent structure: a fully connected layer, followed by batch normalization, and a Gated Linear Unit (GLU) activation function. The final output from this component undergoes transformation through a Rectified Linear Unit (ReLU) activation function.

The Feature Selection module receives transformed features and processes them through an Attentive Transformer followed by a masking operation. This mechanism facilitates soft feature selection—allowing the model to assign continuous importance weights rather than making binary feature inclusion decisions.

The Attentive Transformer incorporates a fully connected layer, batch normalization, and a Sparsemax normalization function. A critical aspect of this module is its utilization of prior scales, which track cumulative feature usage across previous *Steps*. This temporal feature utilization tracking enables the model to distribute attention across complementary features throughout the decision process.

The resulting *Mask* is instrumental in focusing the model’s attention on the most informative features for each specific input. Additionally, these masks serve as powerful tools for model explainability—enabling quantitative assessment of feature importance across individual predictions and aggregate decision patterns. Feature importance can be directly quantified by analyzing the magnitude of mask values

3. Background

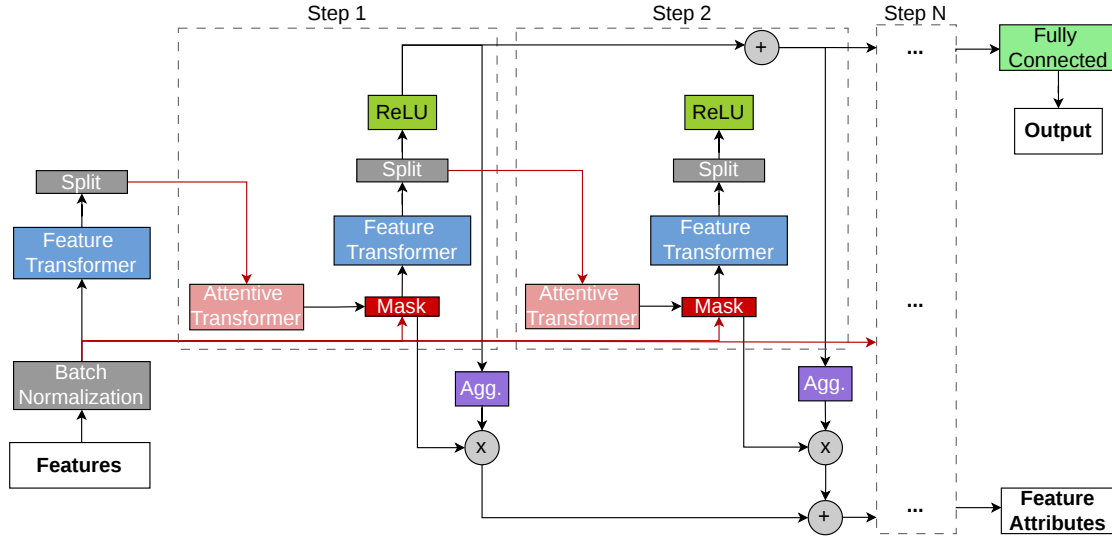


Figure 3.3: TabNet model architecture.

assigned to each feature dimension.

TabNet further enhances its learning capacity through several techniques. It employs residual connections between *Steps* to facilitate gradient flow during training. The architecture also implements a trainable relaxation parameter that controls the trade-off between model capacity and interpretability. Furthermore, TabNet incorporates an entropy-based regularization term that promotes sparsity in feature selection, encouraging the model to focus on minimal yet sufficient feature subsets for accurate predictions.

The training process utilizes a two-phase approach: pretraining with unsupervised reconstruction objectives followed by supervised fine-tuning. This approach leverages TabNet’s inherent ability to capture feature relationships even without labeled data. Empirical evaluations have demonstrated TabNet’s competitive performance against gradient boosting and other deep learning approaches across diverse tabular datasets, while maintaining significantly better interpretability characteristics.

In conclusion, TabNet effectively accomplishes dual objectives: accurate output mapping and transparent feature selection, making it particularly valuable for applications where model decisions require justification or regulatory compliance.

3.4 Ensemble Methodology

Ensemble modeling combines multiple individual models (weak or base learners) to create a robust and accurate prediction system [55]. The effectiveness of ensemble methods stems from their ability to leverage the collective advantages of different models through three key aspects: **model diversity**, which ensures complementary perspectives; **performance benefits**, which demonstrate measurable improvements over single-model approaches; and the **prediction workflow**, which encompasses input processing, model integration through voting mechanisms, and output generation.

Model Diversity: The ensemble consists of multiple independent models working in parallel. Each weak model provides a unique perspective using different algorithms or training parameters, capturing various aspects of the underlying patterns while reducing individual model biases.

Performance Benefits: Ensemble methods demonstrate superior performance due to:

- **Error Reduction:** Different models make different types of errors, which can be averaged out.
- **Robustness:** Less sensitivity to outliers and noise, as incorrect predictions from individual models are mitigated.
- **Generalization:** Diverse models better generalize across different input data and problem scenarios.

Prediction Workflow: The process of ensembling can be represented by the following workflow (Figure 3.4):

1. **Input Processing:** Test input is simultaneously fed into all models, allowing each to process and analyze the same data independently and generate predictions based on unique characteristics and learned patterns.
2. **Model Integration through Voting:** Individual predictions are combined through voting mechanisms:

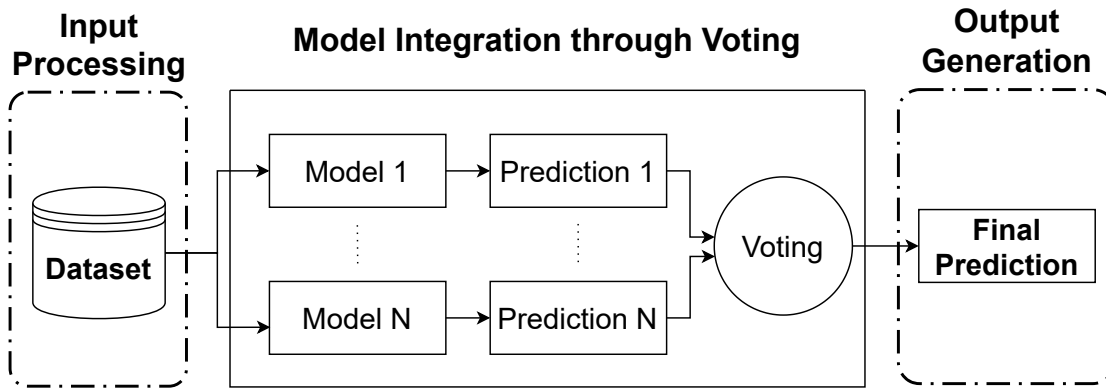


Figure 3.4: Theoretical model ensembling process.

- **Majority Voting:** Each model casts a vote, with the most frequent prediction becoming the final output.
- **Weighted Voting:** Models receive different weights based on historical performance or confidence levels.
- **Soft Voting:** Models provide probability distributions across possible outcomes, which are aggregated for the final prediction.

Output Generation: The voting mechanism aggregates individual predictions to generate a consolidated output, producing more stable and accurate predictions than any single model could independently achieve.

3.4.1 Ensemble Evaluation Metrics

Evaluation of model performance is critical for assessing ML methodologies' efficacy and reliability. Five key performance indicators are used: accuracy, precision, recall, F1-Score, and Receiver Operating Characteristic Area Under the Curve (ROC AUC) score. These metrics provide a multifaceted analysis of predictive capabilities, capturing different aspects of classification effectiveness, including overall correctness, balance between false positives and false negatives, and discriminative power across classification thresholds. Overall, each metric addresses distinct challenges in performance optimization.

3. Background

Accuracy represents the proportion of correct predictions among all predictions made, providing a straightforward measure of overall performance.

$$\text{Binary Classif. Acc.} = \frac{TP + TN}{TP + TN + FP + FN} \quad (3.6)$$

where TP denotes True Positives, FP False Positives, TN True Negatives, and FN False Negatives.

For multiclass classification, accuracy is formulated as:

$$\text{Multi. Classif. Acc.} = \frac{\text{Correct Classifs.}}{\text{All Classifs.}} \quad (3.7)$$

E.g.: With an accuracy of 97.65%, the model can correctly classify approximately nearly all code-regions in the benchmark.

Accuracy quantifies overall correctness but may be misleading in imbalanced scenarios (e.g., when the optimal configuration occurs in only 15% of cases). Nevertheless, it provides a baseline measure of general performance. To obtain a more reliable estimate of generalization, we adopted K-fold cross-validation with $k = 5$ balancing statistical robustness and computational feasibility. Cross-validation accuracy is defined as the mean accuracy across k validation folds, offering an estimate that is less sensitive to data partitioning.

Precision measures the proportion of correctly identified positive predictions among all positive predictions, quantifying the reliability of positive predictions. High precision indicates a low false positive rate.

$$\text{Precision} = \frac{TP}{TP + FP} \quad (3.8)$$

E.g.: A precision of 96.44% indicates that when the model classified a code-region, it was correct 96.44% of the time, resulting in very few false positive predictions, thus very few code-regions are wrongly classified.

Precision is relevant because false positives are costly. Recommending a suboptimal configuration can waste significant runtime. High precision ensures that when

3. Background

the model recommends a configuration, it is indeed the optimal or near-optimal configuration.

Recall (also known as Sensitivity or True Positive Rate) measures the proportion of actual positive cases correctly identified, indicating the model’s ability to find all relevant cases.

$$\text{Recall} = \frac{TP}{TP + FN} \quad (3.9)$$

E.g.: With a recall of 95.56%, the model successfully classified 95.56% of all code-regions. Thus the model successfully finds and correctly classifies most code-regions of each type and very few are misclassified.

Recall addresses the complementary risk of missing optimal configurations. In tuning scenarios, low recall means the model overlooks optimal configurations, leaving potential performance gains undiscovered.

F1-Score provides a balanced measure between precision and recall by calculating their harmonic mean, penalizing extreme values in either metric. A high F1-Score means the model is accurate in number of threads optimization and comprehensive in identifying different code patterns requiring optimization, ensuring high precision is not achieved by only making predictions in high-confidence scenarios while avoiding complex ones.

$$F1 = 2 \cdot \frac{\text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}} \quad (3.10)$$

E.g.: The high F1-Score of 95.96% demonstrates that the model accurately identifies different code-region types with excellent balance between precision and recall. This indicates the model both correctly classifies most code-regions as their true types (low FN) and avoids misclassifying regions into wrong categories (low FP).

In summary, the F1-Score balances precision and recall, which is essential because neither metric alone suffices. A model with high precision but low recall might only predict in obvious cases (e.g., always recommending maximum threads) while missing subtler optimal configurations. The F1-Score penalizes such behavior, encouraging both correctness and coverage.

3. Background

ROC AUC measures the model’s ability to discriminate between classes across various classification thresholds. A score closer to 1.0 indicates better model discrimination capability.

$$\text{ROC AUC} = \int_0^1 \text{TPR}(\text{FPR}^{-1}(t))dt, \quad (3.11)$$

where TPR is the True Positive Rate (Recall), FPR is the False Positive Rate, and t represents a threshold parameter varying from 0 to 1. A high ROC AUC indicates that the model can effectively differentiate between code-regions requiring different numbers of threads, reliably distinguishing between a parallel loop performing best with eight threads and a memory-bound operation working better with four threads.

E.g.: A ROC AUC score of 99.96% demonstrates exceptional discriminative capability of code-regions. This score indicates the model can reliably distinguish between code-regions with minimal misclassification.

ROC AUC measures discriminative ability across thresholds, it is crucial when suboptimal choices incur varying penalties. E.g., choosing 8 instead of 16 threads may cause only a 10% slowdown, while choosing 2 may cause an 80% slowdown. ROC AUC reflects the model’s ability to rank configurations appropriately, not just classify them as optimal or not.

3.5 Learning Outcomes

The previous work by Alcaraz et al. demonstrated that HwPCs can effectively characterize parallel code-regions and that ML models can predict optimal thread configurations. However, manual HwPC reduction does not scale, and using the same HwPCs for both code-region identification and TP optimization negatively impacts prediction accuracy. These insights directly shaped the design decisions of the methodology presented in this thesis.

The dataset construction methodology highlighted the importance of using PAPI preset events to ensure cross-architecture generalizability, and that problem sizes

3. Background

must be carefully chosen to stress different levels of the memory hierarchy, as performance characteristics captured by HwPCs vary significantly depending on memory access patterns.

Regarding the ML algorithms, each captures different aspects of the data: Logistic Regression with Elastic Net performs implicit feature selection, Random Forest reduces variance through parallel bagging, XGBoost iteratively corrects errors through sequential boosting, and TabNet provides interpretable feature selection through attention mechanisms. No single algorithm is universally superior, which motivates combining them into an ensemble.

The ensemble methodology confirmed that combining multiple diverse models through voting mechanisms reduces individual model errors and improves generalization. The evaluation metrics — accuracy, precision, recall, F1-Score, and ROC AUC — each capture different aspects of classification performance, and together provide a comprehensive assessment of model effectiveness.

3. Background

Chapter 4

Methodology For Hardware Performance Counters Reduction and Automatic Tuning using ML Techniques

This chapter addresses a fundamental challenge in performance optimization: how to effectively leverage HwPCs for automatic tuning of parallel applications. Modern processors provide hundreds of available HwPCs, yet determining which ones are truly informative for performance analysis remains a manual and architecture-dependent task. This work tackles this challenge through a data-driven approach that automatically discovers the most relevant HwPCs for different optimization objectives.

The methodology operates through two distinct phases, each requiring different HwPC subsets. The first phase identifies executing code-regions—distinct computational blocks with consistent performance characteristics—using HwPC data. The second phase optimizes Tuning Parameters (TPs) such as number of threads, thread affinity policies, scheduling strategies, and chunk sizes for each identified code-region. Critically, the HwPCs most informative for distinguishing

code-regions differ from those that best predict optimal TP configurations. Therefore, this work develops separate minimal HwPC subsets for each task, reducing monitoring overhead while maintaining predictive accuracy.

The chapter is organized as follows. Section 4.1 examines the use of HwPCs for parallel application tuning, discussing both their benefits and the practical challenges they present, including the large number of available HwPCs and the difficulty of interpreting their relationship to performance outcomes. To address these challenges, it introduces the Machine Learning ensemble methodology, explaining how four complementary algorithms—Logistic Regression with Elastic Net, Random Forest, XGBoost, and TabNet—work together to extract meaningful patterns from HwPC measurements.

Section 4.2 describes the experimental infrastructure and systematic construction of a comprehensive training dataset. This includes characterizing the target hardware architecture, determining compatible HwPC groups that respect hardware monitoring constraints, calculating problem sizes that stress different memory hierarchy levels, and defining OpenMP parameter configurations that explore the performance space.

Section 4.3 provides an overview of the complete methodology, illustrating how the HwPC reduction phase and application execution phase work together to enable automatic performance tuning.

Section 4.4 details the automatic HwPC reduction methodology that identifies minimal counter subsets while preserving accuracy. Through ensemble training, importance score extraction, and reduction using the kneedle algorithm, the methodology produces a single minimal HwPC list for code-region identification and region-specific lists for each TP optimization task.

Finally, Section 4.5 presents the application execution phase, demonstrating how the reduced HwPC sets and trained models enable runtime code-region identification and TP optimization with minimal overhead. This establishes the technical foundation for the automatic performance tuning methodology developed throughout this thesis.

4.1 HwPC Data and Machine Learning

This section examines the use of HwPCs for parallel application tuning and the challenges that arise from their sheer volume and complex interdependencies, which make manual analysis impractical. ML offers a principled way to navigate this complexity, automatically identifying which HwPCs carry the most relevant information for each optimization task. To this end, an ensemble of four complementary algorithms is employed — Logistic Regression with Elastic Net, Random Forest, XGBoost, and TabNet — each contributing distinct strengths: linear relationship modeling, nonlinear pattern recognition, handling of imbalanced classes, and interpretable feature selection for high-dimensional tabular data.

4.1.1 HwPCs and Performance Optimization

HwPCs contain valuable information that can be leveraged for parallel application tuning, as they provide detailed insights into application behavior at the hardware level. By monitoring HwPCs such as cache misses, memory bandwidth utilization, instruction throughput, and branch mispredictions, HwPCs enable the identification of performance bottlenecks and optimization opportunities.

However, effectively utilizing this information presents significant challenges. Modern processors expose hundreds of HwPCs. Furthermore, hardware limitations prevent simultaneous monitoring of all these HwPCs, requiring careful selection of which events to track. Moreover, the complex relationships between HwPCs and performance outcomes make manual interpretation difficult and time-consuming, particularly when attempting to understand how these metrics relate to TPs such as number of threads, scheduling policies, and memory allocation strategies.

The optimization process itself comprises two distinct phases, each with specific monitoring requirements. The first phase involves collecting HwPC data to identify the executing code-region. This identification must occur with minimal overhead, as excessive monitoring can degrade application performance. The second phase focuses on TP optimization: once the code-region is identified, the system must determine which TPs to adjust to improve performance. Critically, these two phases

require different sets of HwPCs—the HwPCs most informative for distinguishing between code-regions may not be the same as those that best predict optimal TP configurations. If either HwPC set is too large, the monitoring and analysis overhead becomes prohibitive, significantly degrading the quality and timeliness of performance predictions. Collecting numerous HwPCs introduces latency in both data gathering and model inference, directly impacting the responsiveness of the tuning system. Therefore, reducing both HwPC sets to their minimal subsets is essential for maintaining low overhead while preserving accuracy.

ML methodologies offer a promising solution to these challenges due to their proven capability in analyzing vast amounts of high-dimensional data. These techniques can automatically identify patterns and relationships that would be impractical to detect manually. Specifically, ML approaches can address two critical aspects of HwPC-based performance optimization: first, they can determine which subset of HwPCs is most informative for accurately characterizing different code-regions; second, they can learn the relationships between HwPC measurements and optimal TP configurations, enabling automated performance tuning across diverse architectures.

4.1.2 Ensemble Algorithms

This section focuses on the ML algorithms used to form the ensemble for the reduction process. To create a robust automatic methodology, an ensemble that combines multiple models trained simultaneously on the same dataset is employed (as presented in Chapter 3). Such ensembles achieve superior performance compared with individual models [55] by leveraging the complementary strengths of different algorithms—some excel at capturing linear relationships while others handle non-linear patterns more effectively, some are more robust to noise while others provide better generalization. Their effectiveness stems from reduced bias and variance through model complementarity, the integration of diverse analytical approaches, enhanced robustness to individual model failures, and improved generalization through model diversity [59]. By integrating predictions from these diverse models, the ensemble achieves more accurate and reliable performance optimization

than any single model alone.

The models for this ensemble were selected to maximize complementarity across different algorithmic paradigms specifically for HwPC feature spaces: linear methods with regularization, tree-based ensembles (both bagging and boosting approaches), and deep learning architectures.

Four models were selected from nine initial candidates through evaluation based on computational efficiency during training and predictive performance metrics. Models with excessive training requirements or inferior accuracy, precision, and recall were discarded to ensure practical deployment in resource-constrained environments.

By combining models with fundamentally different inductive biases that demonstrated high performance across our evaluation criteria, we created a more robust prediction framework that could be generalized across the heterogeneous code-region landscape. This diversity enables handling different aspects of the HwPC feature space: from linear to complex nonlinear relationships, from balanced to imbalanced classes, and from simple to hierarchical decision boundaries. The implementation uses four complementary classifier models, each selected for specific strengths in the feature-target relationship analysis.

Logistic Model with Elastic Net is a combination of the Logistic Model [17] with Elastic Net [80], which integrates both L1 (Lasso, or Least Absolute Shrinkage Selector Operator Regression) [67] and L2 (Ridge) [33] regularizations. This combination offers advantages in managing multicollinearity among predictors, mitigating overfitting, and facilitating feature selection. The model is computationally efficient and effective in scenarios where clear class separation and simple relationships exist between the features and target variables.

In the context of HwPC data, the model addresses the multicollinearity inherent in such measurements (e.g., correlated cache miss events) while providing feature selection through L1 regularization. This is crucial because most HwPCs measure related architectural events. It excels when linear relationships exist between HwPC combinations and performance outcomes, particularly for compute-bound

code-regions where instruction counts linearly predict execution time.

Random Forest [11] is a flexible meta-estimator that can be used for both regression and classification. It is based on Bagging (Bootstrap Aggregation), is resilient to outliers, and is most effective for datasets with a small to medium number of features. It is mainly used to capture the nonlinear relationships between them.

First, the model trains a number of decision trees on various random subsamples of the dataset, called Bootstrap Samples (or bags). Second, it produces the output by averaging (aggregating) individual outputs, thus improving the predictive accuracy and controlling overfitting.

In the context of HwPC data, the model addresses nonlinear interactions in memory-intensive code-regions, where relationships between cache misses, memory bandwidth, and performance are interdependent. Its resilience to outliers is essential given that HwPC values can span several orders of magnitude. Its feature importance mechanism also provides interpretability for identifying which architectural events most influence performance.

XGBoost (Extreme Gradient Boosting) [16] is a fast and scalable implementation of gradient-boosted trees. It reduces the tree search space using the information provided by the distribution of the features. This approach can be used for both regression and classification by handling a wide range of data sizes. This is effective for datasets with unbalanced class distributions. It is based on Boosting, an iterative methodology that sequentially builds models, assesses their errors, and assigns higher weights to incorrectly predicted values to improve subsequent models. As each new model aims to rectify the errors of the previous model, the result is the model with the highest accuracy.

In the context of HwPC data, the model addresses class imbalances that may arise in the dataset (where optimal configurations may be rare) through its weighting mechanisms. Its boosting mechanism captures dependencies between HwPC events and provides strong performance in high-dimensional, sparse feature spaces.

TabNet [7] is a Deep Neural Network designed for tabular data capable of per-

forming both classification and regression tasks. It is particularly effective in scenarios where the dataset has a large number of features with complex relationships with the target variable. It employs a learner that resembles decision-tree-based models, aiming to achieve interpretability and sparse feature selection. TabNet performs both output mapping and feature selection, making it a powerful tool for analyzing and understanding complex tabular datasets.

In the context of HwPC data, the model handles the heterogeneous nature of HwPC data (mixing instruction counts, temporal cycles, and throughput rates) through its attention mechanism, automatically learning which HwPC groups are relevant for specific code-regions. Unlike other models, it performs simultaneous feature selection and prediction, which is crucial when dealing with 50+ HwPCs where many of them may be irrelevant.

4.2 Dataset Construction

The construction of a high-quality dataset is essential for training ML models capable of accurate code-region identification and TP optimization. This dataset must comprehensively capture the characteristics of code-regions across diverse execution conditions, including variations in problem sizes, threading configurations, and OpenMP TPs. Without such coverage, the trained models would lack the data needed to generalize effectively to real-world applications.

This Section describes the concrete construction of the dataset used to train the ML ensemble, applying the dataset creation methodology presented in Chapter 3. Building datasets that are consistent and complete requires the following five steps: (1) characterizing the target hardware architecture to understand available resources and constraints, (2) identifying compatible groups of preset HwPCs that can be monitored simultaneously while respecting hardware limitations, (3) defining problem sizes that systematically stress different levels of the memory hierarchy, from L1 cache through main memory, (4) targeting different OpenMP TP configurations, and (5) creating an execution strategy to generate the dataset.

4.2.1 Hardware Architecture

The hardware architecture analysis was conducted using two specialized tools: the Portable Hardware Locality (hwloc) [12] software package and LIKWID [68]. Hwloc is a widely-used software package that provides a portable abstraction of the hierarchical topology of modern architectures across different operating systems, versions, and hardware configurations. It allows for systematic discovery and representation of computing resources, including processing units, memory hierarchies, and interconnects. The tool exports the topology information through various interfaces, enabling detailed examination of cache structures, memory domains, and processor organization regardless of the underlying hardware vendor or system configuration.

LIKWID (Like I Knew What I'm Doing) is a comprehensive tool suite consisting of command line applications and a library specifically designed for performance-oriented programs. It can be used across a wide range of processors and architectures from different vendors, including Intel, AMD, and ARM. LIKWID provides multiple utilities for architecture characterization, performance measurement, and hardware monitoring. In this work, the `likwid-topology` command was employed to examine the thread affinity, cache hierarchy, and NUMA (Non-Uniform Memory Access) topology of the system. This tool provides detailed information about the physical and logical organization of cores, cache sharing patterns, and memory domain distribution.

Using these tools, the architecture details of the Intel processor system were obtained. The system consists of a Xeon E5-4620 processor composed of 4 sockets, with 8 cores per socket, and 2 threads per core. The memory hierarchy comprises a 32KB L1 data cache and a 256KB L2 cache for each core, and a shared 16MB L3 cache in each processor. The total amount of main memory is 126GB. A summary of the architecture is shown in Table 4.1.

4.2.2 Preset Events per System

After introducing the system used in the experimentation, the next step involves HwPC data collection. For HwPC collection, Performance Application Program-

Table 4.1: Hardware architecture details.

Architecture	Xeon E5-4620
Sockets	4
Cores/Socket	8
Threads/Core	2
L1 Cache	32KB
L2 Cache	256KB
L3 Cache	16MB
Main Mem.	128GB

ming Interface (PAPI) [52, 44] was used.

PAPI provides a standardized set of preset events for performance monitoring across different processor architectures. These preset events are hardware-agnostic abstractions of low-level processor events, allowing for portable performance analysis regardless of the underlying hardware vendor. Examples of preset events include cache misses at different levels (L1, L2, L3), branch instructions and mispredictions, TLB misses, etc.

However, hardware limitations restrict the number of HwPCs that can be monitored simultaneously. Current processors include a large number of available HwPCs, but only a subset can be read at the same time due to the limited number of special-purpose registers built into the processor. For instance, while the Xeon E5-4620 reports 50 available preset events, only a few can be measured concurrently.

To overcome this limitation, the preset events were grouped to maximize the information gathered during each execution while respecting these hardware constraints [4]. The grouping strategy involves creating compatible sets of events that can be measured simultaneously. Each group is measured through a separate execution of the application, ensuring accurate measurements without the overhead and precision loss associated with multiplexing approaches.

The approach of executing the application multiple times—once for each group of HwPCs—provides higher measurement accuracy and complete, reliable performance data. Each execution measures a specific group of compatible HwPCs without interference, ensuring precise measurements essential for training robust ML

models. While this requires multiple executions per configuration, the improved data quality justifies the additional execution time.

The alternative approach of HwPCs multiplexing applies a time-sharing approach, where different groups of HwPCs are measured in different time intervals throughout a single execution. While multiplexing has the advantage of requiring only one execution of the application, it introduces overhead due to HwPC swapping and reduces measurement precision since final HwPC values must be estimated from partial measurements collected in each time interval. These limitations make multiplexing unsuitable when high-accuracy performance data is required for ML model training.

To obtain the available PAPI preset and user-defined events, the command `papi_avail -a` was executed. In the case of the Xeon E5-4620, PAPI reports 50 preset events available in the processor.

The measurable events for the processor were then classified into different event types. The Branches category refers to the conditional branch instructions. The L1, L2, and L3 cache categories refer to Level 1, 2, and 3 data or instruction cache accesses, reads, hits, misses, etc. The TLB preset events refer to data or instruction translation lookaside buffer misses. The Cycles category is a broad one, ranging from cycles with no instruction issue to total cycles. The Operations and Instructions categories refer to the events measuring floating-point operations and instructions. The target architecture's available preset HwPCs are shown in Table 4.2.

Table 4.2: Preset HwPCs available on the Xeon E5-4620 platform.

Type	Xeon E5-4620
Branches	6
L1 cache	5
L2 cache	14
L3 cache	9
TLB	2
Cycles	3
Operations	3
Instructions	8
Total	50

Following the classification, compatible groups for the events were determined through a semi-automated process. The command `papi_event_chooser PRESET [EVENT]` reports which events are compatible with a given event or list of events, using the same output format as `papi_avail`. When incompatible events are selected, the following error is returned: `Event [EVENT] can't be counted with others -1`. Through this process, 12 different compatible groups were identified for the E5-4620 system (Intel), as shown in Table 4.3.

Table 4.3: Compatible PAPI preset event groups for the Intel E5-4620 system.

Group 1	Group 2	Group 3	Group 4
PAPI_L1_DCM	PAPI_L3_TCM	PAPI_TLB_DM	PAPI_STL_ICY
PAPI_L1_ICM	PAPI_L1_LDM	PAPI_TLB_IM	PAPI_BR_UCN
PAPI_L2_DCM	PAPI_L2_STM	PAPI_L1_STM	PAPI_BR_CN
PAPI_L2_ICM	PAPI_L2_DCR	PAPI_L2_DCW	PAPI_BR_TKN
PAPI_L1_TCM	PAPI_L3_DCW		PAPI_BR_NTK
PAPI_L2_TCM	PAPI_L2_ICA		PAPI_BR_INS
PAPI_TOT_INS	PAPI_L2_ICR		
PAPI_L2_DCA	PAPI_L2_TCR		
PAPI_L3_DCA	PAPI_L3_TCW		
PAPI_L3_ICA	PAPI_REF_CYC		

Group 5	Group 6	Group 7	Group 8
PAPI_BR_MSP	PAPI_FP_INS	PAPI_LD_INS	PAPI_L2_DCH
PAPI_BR_PRC	PAPI_L2_ICH	PAPI_SR_INS	PAPI_L3_ICR
PAPI_TOT_CYC	PAPI_FP_OPS		PAPI_L3_TCA
PAPI_L3_DCR			

Group 9	Group 10	Group 11	Group 12
PAPI_L2_TCA	PAPI_L3_TCR	PAPI_SP_OPS	PAPI_DP_OPS
PAPI_L2_TCW	PAPI_VEC_SP		PAPI_VEC_DP
PAPI_FDV_INS			

4.2.3 Problem Size Calculation

After determining the correct groups of events for the architecture, the problem sizes must be determined. Problem sizes were calculated proportionally to memory capacity: for private caches (L1, L2), sizes ranged from one cache unit to the accumulated capacity across all cores in one socket; for shared caches (L3), sizes exceeded lower-level capacity but remained below the shared cache maximum; for main memory, sizes began above the last-level cache and reached 1.5 times the aggregated L3 capacity across all processors. For the Xeon E5-4620 system, the

following formulas were used to perform the calculations.

The problem sizes to stress the L1 Cache were computed with this equation:

$$L_{1n} = 32KiB \times n, \quad (4.1)$$

n denotes the number of cores being stressed (iterations per memory level), where n is $1 \leq n \leq 8$. This yielded a minimum size of 32KiB (single core) and maximum size of 256KiB (all 8 cores in one socket).

For the problem sizes to stress the L2 Cache, we used this equation:

$$L_{2n} = 256KiB \times n, \quad (4.2)$$

where n is $1 \leq n \leq 8$. The range of the L2 problem sizes is a minimum size of 256KiB (single core), and a maximum size of 2MiB (all 8 cores in one socket), just under the L3 limit.

The problem sizes to stress the L3 Cache were computed using this equation, designed to reach the upper limit across the iterations:

$$L_{3n} = 2048KiB + 512KiB + 1902KiB \times (n - 1), \quad \text{where } 1 \leq n \leq 8 \quad (4.3)$$

The minimum size of 2.5MiB (2048KiB + 512KiB) was just above the cumulative L2 capacity of 2MiB across all 8 cores in one socket. The increment of 1902 KiB per iteration distributed the problem sizes evenly across the L3 range, with the maximum size reaching 15.5 MiB, just below the L3 capacity of 16 MB per socket.

Finally, for the main memory, the sizes were computed with the following equation:

$$M_n = 16.5MiB + 20MiB \times (n - 1), \quad \text{where } 1 \leq n \leq 5 \quad (4.4)$$

For the main memory, the number of problem sizes was proportional to the number of processors plus one, resulting in 5 sizes. The minimum problem size was 16.5MiB, just above one processor's L3 cache. The increment of 20MiB per iteration ensured even distribution across the main memory range, with the maximum

size reaching 96.5MiB at $n = 5$, which represented $1.5\times$ the aggregated L3 capacity of all 4 processors ($1.5 \times 64\text{MiB}$).

The summary of the results can be seen in Table 4.4. Note that L3 sizes remained below 16 MB to avoid main memory spillover, while main memory sizes began above 16 MB to ensure main memory utilization. In total, 29 problem sizes were obtained: 8 per cache level determined by the number of cores per socket, and 5 for main memory determined by the number of processors plus one. The step size at each level reflected the corresponding cache unit: 32 KiB for L1, 256 KiB for L2, and 1.9 MiB for L3, ensuring even distribution across each cache range. For main memory, the step of 20 MiB distributed the sizes evenly between the last-level cache capacity and 1.5 times the aggregated L3 capacity across all processors.

Table 4.4: Problem sizes calculated for different memory levels in Xeon E5-4620.

Sizes	Xeon E5-4620	Step	Num. of Sizes
L1 Cache	32 - 256 (KiB)	32 KiB	8
L2 Cache	256 - 2048 (KiB)	256 KiB	8
L3 Cache	2.5 - 15.5 (MiB)	1.9 MiB	8
Main Mem.	16.5 - 96.5 (MiB)	20 MiB	5

4.2.4 OpenMP Tuning Parameters

The methodology is evaluated using a comprehensive dataset of 18 OpenMP code-regions, comprising four STREAM [46] benchmark kernels (**C**opy, **S**cale, **S**um, and **T**riad) with distinct memory access patterns, twelve PolyBench [79, 78] synthetic benchmarks representing common computational patterns in scientific and engineering applications, and two additional computational kernels (Collatz sequences and Friendly numbers).

In this work, a *code-region* refers to a distinct computational block within an application that exhibits consistent performance characteristics and can be independently optimized.

Also, in the OpenMP context, code-regions correspond to parallel loops or parallel sections enclosed by `#pragma omp parallel` directives, each characterized by its computational pattern (e.g., memory-bound, compute-bound) which determines

optimal configuration. To explore the performance space and construct comprehensive HwPC datasets for each code-region, four key OpenMP TPs are varied across executions: the number of threads, thread affinity policy, scheduling policy, and chunk size. These TPs directly influence how parallel work is distributed and how threads interact with the memory hierarchy, making them critical for performance optimization.

The following OpenMP parameters are modified across different executions:

1. **Number of Threads:** Executions begin with a single thread, incrementing by one after each execution until reaching the maximum number of threads available in the system.
2. **Thread Affinity Policy:** Code-regions are executed first with a `close` thread affinity policy followed by a `spread` affinity policy. To implement these policies correctly, the `OMP_PLACES` and `OMP_PROC_BIND` environment variables must be properly configured. `OMP_PLACES` represents the smallest execution unit in the environment (typically a hardware thread), while `OMP_PROC_BIND` defines the thread affinity policy for code-regions at the corresponding nested level.
3. **Scheduling Policy:** The scheduling policy for distributing loop iterations among threads is varied by cycling through `static`, `dynamic`, and `guided` runtime options. These policies are paired with the chunk size parameter.
4. **Chunk Size:** The chunk size determines the number of loop iterations assigned to each thread in a single work distribution unit. To explore different load balancing and cache behavior characteristics, three chunk sizes are used for each scheduling policy in the data collection.
 - **Default:** Implementation-defined default (contiguous equal-sized iteration blocks for `static`; 1 for `dynamic`; for `guided`, chunk sizes decrease adaptively with a minimum of 1 in most implementations).
 - **Small chunks:** Computed as $\frac{1\% \text{ of Iterations}}{\text{Threads}}$ to evaluate fine-grained work distribution with higher scheduling overhead but better load balancing.

- **Medium chunks:** Computed as $\frac{10\% \text{ of Iterations}}{\text{Threads}}$ to balance scheduling overhead with cache locality.

The interaction between scheduling policy and chunk size creates distinct performance trade-offs.

For **static** scheduling, the default contiguous block assignment (approximately $\frac{\text{Iterations}}{\text{Threads}}$ per thread) provides excellent cache locality and zero runtime overhead but poor load balancing when iteration costs vary; the 1% chunks enable better load distribution through round-robin assignment at the cost of cache locality, while 10% chunks balance both concerns.

For **dynamic** scheduling, the default chunk size of 1 offers maximum load balancing but incurs high scheduling overhead from frequent queue accesses — as threads must request new work from a central queue after every single iteration; 1% chunks significantly reduce this overhead while maintaining responsive load balancing (100 scheduling events), and 10% chunks further minimize overhead (10 scheduling events) while preserving moderate load balancing capability.

For **guided** scheduling, which adaptively decreases chunk sizes during execution, the default minimum of 1 allows aggressive late-stage load balancing; setting 1% as the minimum reduces scheduling overhead while maintaining adaptivity, and 10% prioritizes cache locality and overhead reduction by preventing chunk sizes from becoming too small.

These chunk size choices explore the spectrum from fine-grained (1%) to coarse-grained (10%) parallelism for each policy. Without prior knowledge of a code-region’s computational characteristics, the optimal configuration cannot be determined *a priori*, motivating this exploration of the TP space.

A summary of the OpenMP TPs targeted in this work and their possible values is shown in Table 4.5, where the combination of thread affinity policy, scheduling policy, and chunk size results in a total of 18 TP configurations per code-region execution. The measurements collected for each code-region and TP configuration

establish the association between TPs and code-regions, which is later exploited during the HwPC reduction phase to identify the most relevant HwPCs for predicting the optimal TP value for each code-region.

Table 4.5: OpenMP TPs and their configuration values for the Xeon E5-4620 platform.

OpenMP Parameter	Config. Values
Number of Threads	1-32
Thread Affinity Policy	close, spread
Scheduling Policy	static, dynamic, guided
Chunk Size	default, $\frac{1\% \text{ of Iterations}}{\text{Threads}}$, $\frac{10\% \text{ of Iterations}}{\text{Threads}}$

4.2.5 Execution Methodology

To characterize the performance behavior of code-regions, HwPC data must be collected under diverse execution conditions to form comprehensive performance signatures. These signatures are constituted by HwPC values that capture diverse performance metrics such as memory accesses at each cache level, instruction counts, branch events, TLB misses, and cycle counts.

Collecting metrics across multiple execution scenarios ensures that the signatures capture the full range of performance characteristics exhibited by each code-region across different computational scenarios. The variation of execution parameters—including HwPC event groups, problem sizes, and OpenMP TP configurations—enables the identification of performance patterns and relationships between code behavior and hardware resource utilization. By executing code-regions across multiple configurations, the methodology generates a robust dataset that accounts for the impact on HwPCs of different number of threads configurations, memory access patterns, and workload distributions.

The execution of code-regions follows a systematic approach to gather HwPC data across various execution conditions:

1. Groups of hardware events (Table 4.3)
2. Problem sizes (Table 4.4)

3. Tunable OpenMP parameters (Table 4.5)

The execution strategy involves multiple repetitions across different configurations that can be calculated as:

$$E = S \times T \times P \times OP \times N, \quad (4.5)$$

where E denotes executions, S HwPC sets, T number of threads, P problem sizes, OP OpenMP TP combinations, and N repetitions.

To ensure statistical significance, each unique configuration is executed 100 times (i.e., $N = 100$). Once the architecture details are established and the values for HwPC sets (S), thread configurations (T), problem sizes (P), and OpenMP thread placements (O) are determined, the total execution count can be computed using equation 4.5. The final dataset undergoes preprocessing, where the HwPC values are normalized, and null values and zero-variance features are removed. This dataset is formally defined as:

$$\mathcal{D} = (\mathbf{l}, \mathbf{h}, \mathbf{t}) \quad (4.6)$$

where $\mathbf{l} \in \mathcal{L}$ represents code-region labels, $\mathbf{h} \in \mathcal{H}$ denotes HwPC values (forming the performance signatures), and $\mathbf{t} \in \mathcal{T}$ represents TP values.

Table 4.6 presents a summary of the execution configurations associated with their particular values for the Xeon E5-4620 platform.

Table 4.6: Number of the execution configurations for a specific platform (Xeon E5-4620).

Parameter		Number of Configs. for Xeon E5-4620
Counter Sets	S	12
Threads	T	32
Problem Sizes	P	29
OpenMP TPs	OP	18
Repetitions	N	100
Total	E	20044800

4.3 Application Performance Tuning using Machine Learning

This section presents an overview of a novel automatic methodology for tuning parallel applications using HwPCs, transforming raw HwPC data into actionable performance insights through ML methodologies. The methodology addresses performance analysis challenges by identifying code-regions and optimizing TPs using HwPC data. The ensemble identifies both minimal sets of HwPCs required for code-region identification and for TP optimization, improving efficiency and accuracy while automating application performance optimization.

Remind that in this work, a *code-region* refers to a distinct computational block within an application that exhibits consistent performance characteristics and can be independently optimized. Also, in an OpenMP context, code-regions typically correspond to parallel loops or parallel sections. For GPU applications, code-regions correspond to individual CUDA kernels. The methodology identifies code-regions through their unique HwPC signatures rather than static code analysis.

A key motivation for using separate HwPC subsets for code-region identification and TP optimization is that the most discriminative HwPCs for recognizing computational patterns could not be the most informative for predicting TPs. For example, cache-related HwPCs can distinguish compute- from memory-bound code-regions, while instruction throughput HwPCs can be related to core utilization and thread placement TPs. By decoupling the subsets, the methodology maximizes predictive accuracy for both tasks.

The proposed methodology is composed of two phases, that are presented in Figure 4.1: a **HwPC Reduction phase**, which is performed off-line only once for a specific architecture using the input dataset, and an **Application Execution phase**, which is applied online to optimize a given application for a given architecture.

The input of the **HwPC Reduction Phase** is a dataset composed of representative code-regions selected to cover the search space, which encompasses the diverse performance behaviors that can arise from different computational patterns, to-

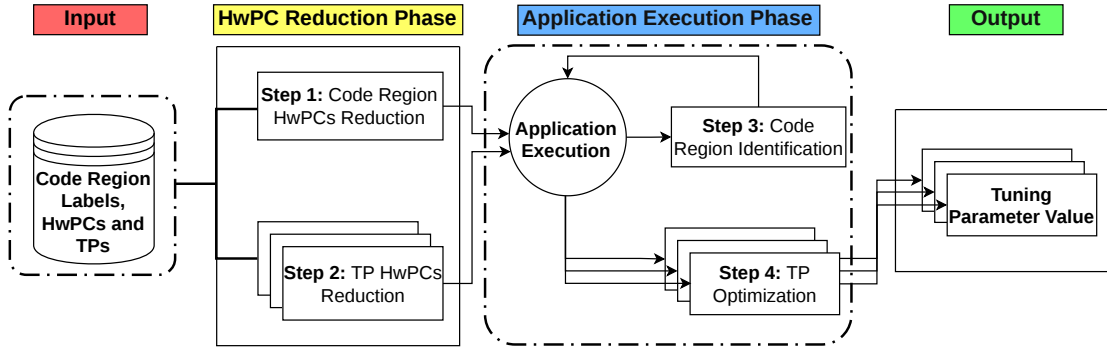


Figure 4.1: Detailed flow of the proposed methodology for automatic tuning of parallel applications.

gether with their associated HwPC values and TP values. Code-regions are chosen to represent a distinct performance pattern, building a balanced dataset from multiple execution scenarios.

This phase’s objectives are to minimize the sets of HwPCs needed to identify code-regions, and optimize TPs. Section 4.4 describes in detail this phase’s two steps.

- **Step 1: Code-region HwPCs Reduction.** The $\mathbf{l}$ and $\mathbf{h}$ components of the dataset are used to identify relevant HwPCs for code-region classification, producing a reduced HwPC list and a classification model.
- **Step 2: Tuning Parameter HwPCs Reduction.** The $\mathbf{l}$, $\mathbf{h}$, and $\mathbf{t}$ components of the dataset are used to identify relevant HwPCs for each TP of each code-region, where the association between TPs and code-regions is established during dataset construction by measuring the performance impact of each TP configuration on each code-region. This results in a reduced HwPC list and a prediction model for each specific TP of each code-region.

The **Application Execution Phase** [30] involves the optimization of a given application using the reduced sets of HwPCs obtained in the previous phase. It includes two steps described in detail in Section 4.5.

- **Step 3: Code-region Identification.** At runtime, the values of the HwPCs from the list produced in Step 1 are collected, and the classification

model is used to identify the executing code-region.

- **Step 4: Tuning Parameter Optimization.** After identifying the executing code-region, for each TP associated with this code-region, it collects the values of the HwPCs of the list produced in Step 2, and the corresponding prediction model is used to determine the optimal TP value.

Consequently, the methodology's output is a value $t_{opt} \in \mathcal{T}$ for each TP that optimizes the performance of each code-region in an application.

In practice, as it will be discussed in Chapter 5, the computational overhead of the proposed approach remains modest: the reduction phase is performed offline, while online execution requires monitoring only 4–7 HwPCs per task, adding negligible runtime overhead.

4.4 HwPC Reduction and Model Training

Having constructed the dataset (Section 4.2) and selected the ensemble models (Section 4.1), the HwPC reduction process can be applied to obtain minimal sets of HwPCs for code-region characterization and TP optimization. The methodology determines which HwPCs are most informative for each task, obtaining a minimal set that reduces monitoring overhead.

The dual-purpose approach, as illustrated in Figure 4.2, operates through two branches. The first branch (Step 1) identifies important HwPCs for code-region identification using sub-datasets containing code-region labels and HwPC values. The second branch (Step 2) identifies important HwPCs for optimizing TPs using separate sub-datasets for each code-region and TP combination, with their corresponding HwPCs. These sub-datasets are derived from the complete dataset $\mathcal{D} = (\mathbf{l}, \mathbf{h}, \mathbf{t})$ by filtering relevant components: Step 1 uses only $(\mathbf{l}, \mathbf{h})$ pairs for classification tasks, while Step 2 partitions the data by code-region and creates region-specific subsets containing $(\mathbf{h}, \mathbf{t})$ pairs for each TP being optimized.

Both undergo preprocessing to remove zero-variance features. The methodology employs an ML ensemble through three sequential steps: (1) training and validation

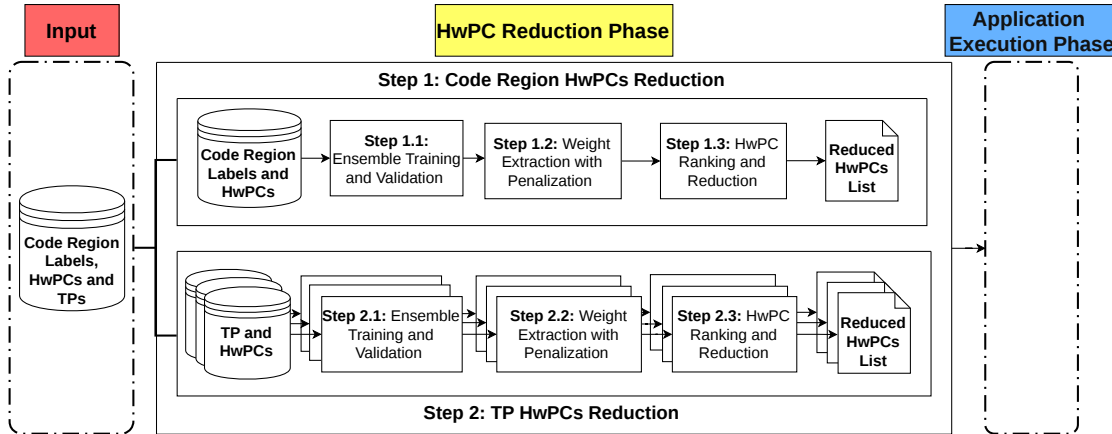


Figure 4.2: HwPCs reduction with model ensembles.

ing classifier models to establish relationships between code-regions and HwPCs (Step 1.1), and between TPs and HwPCs (Step 2.1); (2) extracting and quantifying importance scores for individual HwPCs across all ensemble models for code-region identification (Step 1.2) and TP optimization (Step 2.2); and (3) ranking and reduction to generate a minimal HwPC set for code-region identification (Step 1.3) and minimal specialized sets for TP optimization (Step 2.3). During inference, models generate individual predictions with the final prediction determined through majority voting, enhancing robustness. Model performance is evaluated using accuracy to quantify correct classifications.

4.4.1 Ensemble Training and Validation

To establish robust predictive models capable of identifying minimal informative HwPC subsets, the ensemble must first learn the relationships between HwPCs and their target variables (code-regions or TP values) through training and validation.

The ensemble training and validation correspond to Steps 1.1 and 2.1 in Figure 4.2. The ensemble methodology integrates the four models introduced in Section 4.1.2: Logistic Regression with Elastic Net, Random Forest, XGBoost, and TabNet, capturing linear and nonlinear relationships across varying dataset characteristics. The implementation utilized a 70-30 split, with 30% reserved for final testing to provide an unbiased evaluation of generalization capabilities. The re-

maining 70% underwent 5-fold stratified cross-validation, preserving code-region class distributions across folds. In each iteration, the models are trained on approximately 56% of the total data and validated on 14%. A value of $k = 5$ [10, 31] was chosen to balance statistical reliability and computational cost (5 training iterations). Larger k values (e.g., 10) doubled computational cost with marginal accuracy gains, whereas smaller values (e.g., 3) led to higher variance in validation scores. The stratified approach further ensured consistent handling of imbalanced TP distributions. Remember that accuracy served as the primary evaluation metric throughout.

4.4.2 Weight Extraction

Once the ensemble models are trained, their internal representations must be analyzed to quantify which HwPCs contribute most significantly to accurate predictions, enabling the identification of minimal informative subsets.

The weight extraction process corresponds to Steps 1.2 and 2.2 in Figure 4.2. Following training, each HwPC received normalized importance scores (0.0 to 1.0) based on model-specific criteria. Logistic Model with Elastic Net derives importance from coefficient magnitudes. Random Forest quantifies it through an average impurity reduction. XGBoost evaluates importance based on performance gain. TabNet employs an attention mechanism for feature contributions. The final importance score is computed using the weighted average of the individual model scores. Each model's weight corresponds to its validation accuracy. Figure 4.3 illustrates this process. The final weighted score W_i for the i th HwPC is calculated as:

$$W_i = \frac{\sum_{j=0}^3 c_{i,j} w_j}{\sum_{j=0}^3 w_j}, \quad (4.7)$$

where $c_{i,j}$ is a score of the i th HwPC from the j th model and w_j is an accuracy-based weight of the j th model.

This formulation ensures that models with higher validation accuracy contribute

Log. Reg.		Rand. For.		XGBoost		TabNet		Wght. Extr.	
HwPC	Score	HwPC	Score	HwPC	Score	HwPC	Score	HwPC	W. Avg.
0	0.001	0	0.020	0	0.028	0	0.014	0	0.016
1	0.368	1	0.259	1	0.033	1	0.313	1	0.239
2	0.373	2	0.389	2	0.068	2	0.078	2	0.228
...	...	...	...	...	...	...	...	...	...
N-1	0.004	N-1	0.014	N-1	0.027	N-1	0.007	N-1	0.013
Wght.	0.85	Wght.	0.99	Wght.	0.94	Wght.	0.89		

Figure 4.3: Step 2: HwPC importance per model and weighted average.

more significantly to the final importance scores.

4.4.3 HwPC Ranking and Reduction

With importance scores assigned to all HwPCs, the methodology must determine the minimal subset that preserves model accuracy while eliminating redundant or low-contribution HwPCs.

The HwPC ranking and reduction process corresponds to Steps 1.3 and 2.3 in Figure 4.2. The kneedle algorithm [64] was used to select HwPCs for code-region identification and TP optimization. The reduction process ranks HwPCs in descending order based on weighted average scores (Figure 4.4a).

Then, as shown in Figure 4.4b, a reference line is established between points (0,0) and (N-1, accuracy N-1), where N is the total number of HwPCs and accuracy N-1 is the validation accuracy using all HwPCs. The models are trained iteratively, starting with the highest-ranked HwPC and progressively adding HwPCs in rank order.

For each model, the distance between the accuracy and the reference line is calculated. The process stops when adding a new HwPC no longer increases this distance, indicating the *knee point* where accuracy gains become marginal. The minimum effective set of HwPCs includes those added up to this point, balancing model complexity with accuracy. Algorithm 1 summarizes this reduction process.

Algorithm 1 HwPCs Reduction

Input: List H of N HwPCs ordered by rank

Output: Reduced list of K HwPCs

/ The reference line is the one going from the accuracy using 0 HwPCs (0) and the accuracy using all HwPCs (already computed in Step 1) */*

```

1: Compute straight line  $L$  between  $(0, 0)$  and  $(N - 1, Accuracy_{N-1})$ 
2:  $K \leftarrow 1$ 
3:  $OldDiff \leftarrow 0$  // The accuracy with 0 HwPCs
4: Obtain sub-array  $S_1$  of first HwPC from  $H$ 
5: Train and validate an Ensemble using  $S_1$ 
6:  $Acc \leftarrow$  validation accuracy
7:  $NewDiff \leftarrow$  distance between  $(1, Acc)$  and  $L$ 
8: while  $NewDiff > OldDiff$  do
    /* Determining the knee point */
9:      $K++$ 
10:    Obtain sub-array  $S_k$  of first  $k$  HwPCs from  $H$ 
11:    Train and validate an Ensemble using  $S_k$ 
12:     $Acc \leftarrow$  validation accuracy
13:     $OldDiff \leftarrow NewDiff$ 
14:     $NewDiff \leftarrow$  Distance between  $(k, Acc)$  and  $L$ 
15: end while
16:  $Result \leftarrow$  First  $K - 1$  HwPCs from  $H$ 

```

The reduction process produces a minimal list of HwPCs for code-region identification, and TP-specific lists per code-region for TP optimization:

- $HwPC_i = h_0, h_1, \dots, h_{C-1}$: The minimal set of HwPCs to classify the code-region i
- $HwPC_{ij} = h_0, h_1, \dots, h_{T-1}$: The minimal set of HwPCs to tune the j th TP for the i th code-region

where $0 < C, T \leq N - 1$. At the end of the reduction process, the final ensemble models trained with these reduced HwPC sets are stored. These ensembles are then used during the subsequent Application Execution Phase for code-region identification and TP optimization.

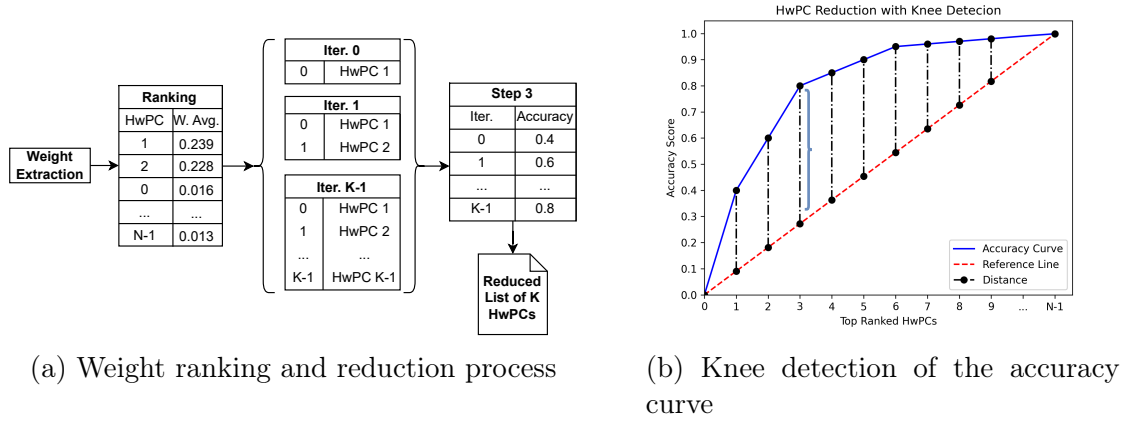


Figure 4.4: Step 3: HwPC reduction considering the knee point.

4.5 Code-Region Identification and Application Execution Phase

After obtaining the reduced list of HwPCs for code-region identification and region-specific lists of HwPCs for TP optimization, the ensemble can identify code-regions and optimize their TP values. The identification and optimization process is performed online and can be used across different executions and applications.

Figure 4.5 delineates the process for code-region identification and TP optimization during application execution. The process begins when the target application is running. First, Step 3 identifies different code-regions within the application. In Step 3.1, the methodology collects measurements from a reduced set of HwPCs that were specifically selected to identify code-regions.

In Step 3.2, this set of HwPCs' values is the input to the code-region Identifier Model generated in the Step 1 of the HwPC reduction phase described in Section 4.4. The final code-region class indicated by the ensemble is decided by majority vote, that is, whichever code-region class receives the most votes.

As observed in Figure 4.5, upon code-region identification, the process transitions to Step 4, TP optimization. In Step 4.1, the system collects a specific set of HwPC measurements from the executing application. Importantly, each identified

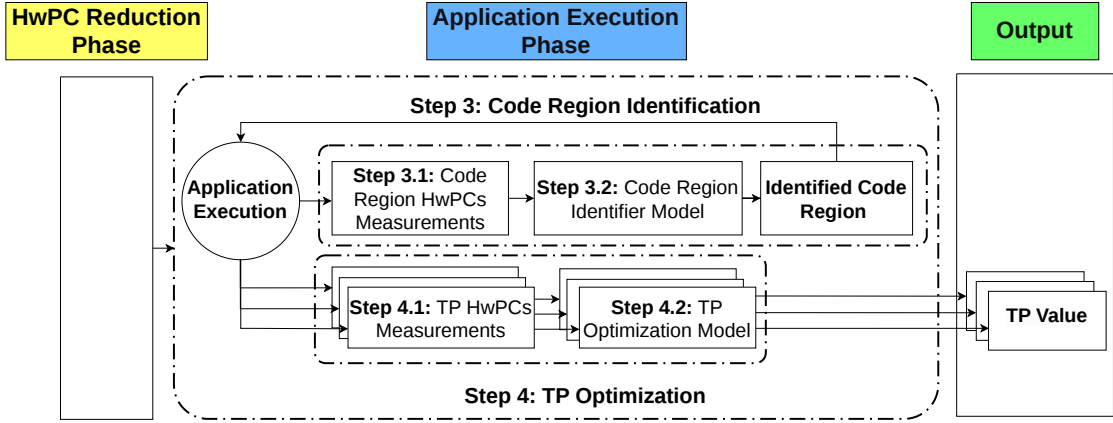


Figure 4.5: Code-region identification and TP optimization with model ensembles.

code-region uses a different set of HwPCs for each TP being optimized. This means that for a given code-region, the system monitors one specific set of HwPCs when optimizing TP A, and a different set of HwPCs when optimizing TP B. This approach allows for detailed analysis of how each individual TP affects performance within that specific code-region.

In Step 4.2, the collected HwPC measurements are passed to the appropriate optimization models. For each code-region, there are multiple TP optimization models - one for each TP. Each of these models was trained using the reduced set of HwPCs that are most relevant for optimizing that specific TP within that particular code-region. As for the code-region identification, each optimization model is an ensemble of the four ML classifiers and the final optimized TP value is determined by majority vote.

Steps 4.1 and 4.2 are then repeated sequentially for each remaining TP associated with the identified code-region, until optimal values have been determined for all TPs. The outcome of this process is a set of highly tailored TP optimizations. Each TP value is generated by a specialized model trained on performance data specific to both the code-region and TP in question.

Chapter 5

Evaluation of the Proposed Methodology

This chapter analyzes the results obtained using the proposed ensemble methodology for HwPC reduction, code-region identification, and TP optimization.

Section 5.1 validates the reliability of the ensemble methodology through a comparative analysis with the findings in our previous work [4]. Section 5.2 continues the evaluation using a dataset consisting of code-regions already known to the models, providing a baseline for performance assessment. Section 5.3 demonstrates the robustness and generalization capabilities of the methodology by extending the evaluation to previously unseen code-regions extracted from the NAS parallel benchmarks (NPB) [8]. Section 5.4 completes the analysis using six GPU architectures and datasets with 5-7 different kernels, demonstrating the methodology’s portability and generalizability.

All CPU benchmarks were compiled using GCC version 9.2.0 with -O2 optimization, and executed on the 32-core Xeon E5-4620 platform and the 12-core Xeon E5645. GPU benchmarks were compiled with NVCC using CUDA 10.1 (driver 418.67) for GTX 680, GTX 750, GTX 1070 and RTX 2080, CUDA 12.1 (driver 535.183) for RTX 3090, and CUDA 12.3 (driver 545.23) for RTX 4080.

This cross-platform evaluation demonstrates the versatility of the methodology and its consistent effectiveness across both CPU and GPU architectures, establishing its broad applicability in diverse computing environments.

5.1 Comparison with PCA and Correlation-Based Methodology

The results obtained using the ensemble methodology are first compared to those reported in the previous work [4]. The same benchmark, STREAM [46], comprising four code-regions — **Copy**, **Scale**, **Sum**, and **Triad** — and the same hardware, a Xeon E5645 processor, are used for this comparison.

In this architecture, the set of preset HwPCs included the following event types and the corresponding number of HwPCs presented in Table 5.1. As a result of

Table 5.1: Preset HwPCs available on the Xeon E5645 platform.

Type	Xeon E5645
Branches	7
L1 cache	8
L2 cache	16
L3 cache	10
TLB	3
Cycles	3
Operations	3
Instructions	8
Total	58

applying the PCA and correlation-based methodology proposed in [4], the number of HwPCs necessary to classify these patterns was reduced from 58 to 20. However, using the ensemble methodology proposed in this thesis, the number of HwPCs is reduced from 58 to 5, which includes a diverse set of event types presented in Table 5.2. Four of these HwPCs (`PAPI_DP_OPS`, `PAPI_FP_OPS`, `PAPI_BR_TKN`, `PAPI_SR_INS`) overlap with the reduced set from [4], while the fifth, `PAPI_L1_ICH`, exhibits strong correlation (0.9781) with `PAPI_BR_MSP` from that set.

In [4], the reduced set was later used to train an Artificial Neural Network (ANN), which was utilized to predict the parallel code-region class based on the values of

5. Evaluation of the Proposed Methodology

Table 5.2: Reduced HwPCs for code-region identification.

Instructions	Operations	Branches	Cache L1
PAPI_SR_INS	PAPI_FP_OPS	PAPI_BR_TKN	PAPI_L1_ICH
	PAPI_DP_OPS		

Table 5.3: Evaluation results for Xeon E5645.

Results/Methodology	PCA & Correlation	Ensemble 1	Ensemble 2
Reduced Set	20	5	5
Accuracy	99.98%	93.83%	99.20%
K-Fold Accuracy	-	85.94%	96.10%

HwPCs. The training set consisted of 54 problem sizes (432000 entries), while the remaining 2 sizes were reserved for testing (16000 entries). Table 5.3 shows that the ANN accurately classified 99.98% of the test set in this case.

To ensure fair comparison, the proposed methodology was initially evaluated using identical train/test splits as the baseline PCA method. This configuration (Ensemble 1 in Table 5.3) achieved a classification accuracy of 93.83% on the test set. Subsequently, stratified 5-fold cross-validation yielded an average accuracy of 85.94%. However, applying a random stratified 5-fold cross-validation partitioning the dataset into 70% for training and validation and 30% for final testing, the model (Ensemble 2 in the Table 5.3) was able to classify accurately 99.20% of the test set. We also performed a stratified 5 fold cross validation, achieving an average cross validated score of 96.10%. Therefore, the proposed methodology achieves comparable results using only 5 automatically determined HwPCs, minimizing the number of executions over selected HwPC groups to improve measurement precision while keeping overhead low.

5.2 Identification and Tuning of OpenMP Regions with a Comprehensive Dataset

Using the minimal set of HwPCs for code-region identification, and the minimal sets of HwPCs for each TP, we can evaluate the effectiveness of the ensemble methodology. The methodology goes first through identifying the code-regions

5. Evaluation of the Proposed Methodology

with a minimal set of HwPCs. Once the code-region is identified, the methodology proceeds to optimize different TPs.

For comparison, the OpenTuner [6] tool was employed. It is an extensible autotuning tool that leverages multiple search techniques to efficiently explore configuration spaces. Its pluggable architecture enables sophisticated optimization strategies that support various hardware platforms by abstracting configuration spaces and search mechanisms.

5.2.1 Evaluation Dataset

The methodology was tested on an 18 code-regions dataset [29] that was generated following the methodology presented in Section 4.2 (summarized in Table 5.4):

- **STREAM**: Four code-regions— `Copy_E`, `Scale_E`, `Add_E`, and `Triad_E`—each with distinct memory access patterns and operation counts [46]. `Copy_E` performs simple array copying, `Scale_E` multiplies arrays by a scalar, `Add_E` sums two arrays, and `Triad_E` combines addition and scalar multiplication operations.
- **PolyBench**: Twelve code-regions from synthetic benchmarks for common computational programs in scientific and engineering applications [79, 78].
 1. **Stencil operations (3 code-regions)**: `2PStencil_E` (1D two-point stencil), `2D4PStencil_E` (2D four-point stencil), and `Stencil` (3D seven-point stencil). These code-regions represent common computational patterns in scientific computing with neighbor-dependent calculations and varying spatial locality characteristics.
 2. **Matrix operations (3 code-regions)**: `Matxvec` (matrix-vector multiplication), `MatrixMult` (optimized matrix-matrix multiplication), and `MatrixMultNOpt` (non-optimized matrix-matrix multiplication with different memory access patterns). These code-regions stress both cache hierarchy and arithmetic units with different optimization strategies.
 3. **Strided access patterns (5 code-regions)**: `Stride2_E`, `Stride4_E`,

5. Evaluation of the Proposed Methodology

`Stride16_E`, `Stride64_E`, and `Rows`. These code-regions perform array copy operations with different stride lengths (powers of 2) and access patterns, specifically designed to stress cache line utilization and spatial locality at different granularities.

4. **Reduction operations (1 code-region):** `Reduction_E` performs parallel summation across an array using OpenMP reduction, testing thread synchronization and cache coherency mechanisms.

- **Irregular Computation Patterns:** Two code-regions for computing Collatz sequences and Friendly numbers with different computational load per iteration. These code-regions exhibit irregular, data-dependent control flow and workload imbalance across threads, evaluated with different OpenMP scheduling policies (`static`, `dynamic`, `guided`) and chunk sizes to capture diverse parallel execution behaviors.

The HwPC collection was conducted on a 32-core Xeon E5-4620 system consisting of 4 sockets with 8 cores per socket, whose full architecture details are summarized in Table 4.1. The preset HwPCs available for this architecture are shown in Table 5.5. Each HwPC group was measured by PAPI across multiple executions, systematically varying TPs, including number of threads, thread affinity policy, scheduling policy, and chunk size. Problem sizes were computed using the methodology in [5] to stress different memory hierarchy levels. For statistical significance, 100 executions were conducted for each combination of HwPC group, problem size, and configuration. As a result, the dataset created comprises 18 code-regions spanning diverse computational patterns — memory-bound streaming operations, matrix computations, strided access patterns, reduction operations, and irregular control flow — providing a comprehensive basis for evaluating the methodology across a wide range of parallel applications.

5.2.2 Code-Region Identification

This section evaluates the performance of code-region identification using the ensemble methodology presented in Section 4.4. The evaluation employed stratified

5. Evaluation of the Proposed Methodology

Table 5.4: Code-regions grouped by benchmark suite and computational characteristics.

Group	Code-region	Description
STREAM (4 code-regions)	Copy_E	Simple array copy operation ($c[i] = a[i]$)
	Scale_E	Array scaled by constant ($c[i] = \text{scalar} \times a[i]$)
	Add_E	Vector addition ($c[i] = a[i] + b[i]$)
	Triad_E	Combined addition and scaling ($c[i] = a[i] + \text{scalar} \times b[i]$)
PolyBench (12 code-regions)	2PStencil_E	1D two-point stencil with nearest neighbors
	2D4PStencil_E	2D four-point stencil (up, down, left, right neighbors)
	Stencil	3D seven-point stencil operation
	Matxvec	Matrix-vector multiplication
	MatrixMult	Matrix-matrix multiplication with optimized memory access
	MatrixMultNOpt	Matrix-matrix multiplication with non-optimized access pattern
	Stride2_E	Array copy with stride of 2 elements
	Stride4_E	Array copy with stride of 4 elements
	Stride16_E	Array copy with stride of 16 elements
	Stride64_E	Array copy with stride of 64 elements
	Rows_E	Row-major 2D array access pattern
	Reduction_E	Parallel summation with OpenMP reduction
Irregular Computation (2 code-regions)	Collatz	Collatz conjecture with variable OpenMP scheduling and chunk sizes
	Friends	Friendly number calculator with variable OpenMP scheduling and chunk sizes

Table 5.5: Preset HwPCs available on the Xeon E5-4620 platform.

Branches	Cache L1	Cache L2	Cache L3	TLB	Cycles	Ops.	Ins.	Total
6	5	14	9	2	3	3	8	50

5-fold cross-validation, partitioning the dataset into 70% for training and validation and 30% for final testing. As indicated in Table 5.6 the ensemble achieved an impressive average cross-validated score of 97.65%. As a result, the ensemble was able to accurately identify code-regions using a minimal set of HwPCs. The evaluation metrics encompass K-fold accuracy, precision, recall, F1-score, and ROC AUC, alongside the comparison between full and reduced HwPC sets required for accurate identification.

The results demonstrate exceptional performance across all metrics. The K-fold

5. Evaluation of the Proposed Methodology

accuracy of 0.9765 means the ensemble correctly identifies which code-region is executing in 97.65% of cases—missing only about 2-3 predictions out of every 100. The ROC AUC of 0.9996 confirms the methodology’s ability to reliably distinguish between different code-regions, achieving near-perfect separation with minimal confusion between regions. The precision of 0.9644 indicates high reliability: when the ensemble identifies a code-region, it is correct 96.44% of the time, minimizing false positives. The recall of 0.9556 shows strong detection capability: the methodology successfully catches 95.56% of all actual occurrences of each code-region, minimizing false negatives. Together, these balanced metrics yield an F1-score of 0.9596, confirming that the methodology achieves both high accuracy in its predictions and completeness in its detection without compromising either aspect.

Notably, the ensemble achieved this high performance while dramatically reducing the required HwPC set from 50 to only 6 HwPCs—an 88% reduction. This substantial dimensionality reduction, coupled with maintained classification accuracy, validates the effectiveness of the methodology across different benchmark suites and suggests strong generalizability for identifying relevant HwPCs in code-region classification tasks. Starting with the 50 available HwPCs, presented in Chapter 4, the methodology reduced the set to 6 HwPCs, shown in Table 5.7, for code-region identification while maintaining high prediction capabilities, as shown in Table 5.6

Table 5.6: Evaluation results for code-region identification for the expanded benchmark on the Xeon E5-4620.

Metric	K-Fold Accuracy	Precision	Recall	F1-Score	ROC AUC	Full Set	Reduced
Label	0.9765	0.9644	0.9556	0.9596	0.9996	50	6

The methodology further refined the reduction process at the TP level, creating unique HwPC sets for each code-region’s TPs. Starting with 50 available HwPCs, the methodology successfully reduced the set to only 5-6 HwPCs for each TP. As an example, Table 5.8 shows the 5 HwPCs selected for optimizing the number of threads for the `2D4PStencil_E` code-region.

These results reflect a key insight: for code-region classification, the relevant Hw-

5. Evaluation of the Proposed Methodology

Table 5.7: Reduced HwPCs for code-region identification.

Group	HwPC	Description
Instructions	PAPI_SR_INS	Store instructions
	PAPI_FDV_INS	Floating point divide instructions
Operations	PAPI_FP_OPS	Floating point operations
	PAPI_DP_OPS	Double precision FP operations
Branches	PAPI_BR_NTK	Branches not taken
	PAPI_BR_MSP	Branch mispredictions

Table 5.8: Reduced HwPCs for 2D4PStencil_E code-region’s number of threads optimization.

L2 Cache	L3 Cache	Operations
PAPI_L2_ICH	PAPI_L3_TCM	PAPI_FP_OPS
PAPI_L2_DCH		
PAPI_L2_DCR		

PCs capture computational characteristics such as operation counts, instruction mix, and control flow (branches). However, for predicting optimal number of threads, the relevant HwPCs depend on the code-region’s computational profile. For memory-intensive regions like 2D4PStencil_E, the selected HwPCs predominantly focus on cache behavior across different memory hierarchy levels (L2, L3), as shown previously. In contrast, compute-intensive regions require HwPCs that capture arithmetic operations and instruction throughput. Table 5.9 presents an overview of the top HwPCs selected for each code-region, illustrating how different computational patterns require distinct HwPC sets.

5.2.3 Tuning Parameter: Number of Threads

This section evaluates the optimization of the number of threads TP for various code-regions.

The evaluation starts with a focus on the trade-off between performance (time) and resource (core) utilization. To quantify the efficiency of using different numbers of threads, the performance index $Pi(X)$ defined in [15] was calculated using the following equation:

$$Pi(X) = \frac{T_t(X)}{E(X)} = \frac{X \cdot T_t(X)^2}{T_t(1)}, \quad (5.1)$$

5. Evaluation of the Proposed Methodology

Table 5.9: HwPCs used by each code-region for number of threads optimization (grouped by preset event type).

Code-region	Br.	Ops.	Ins.		L1 Cache		L2 Cache				L3 Cache			TLB	Cyc.							
	PAPI_BR_MSP	PAPI_DP_OPS	PAPI_FP_OPS	PAPI_LD_INS	PAPI_SR_INS	PAPI_TOT_INS	PAPI_L1_ICM	PAPI_L1_LDM	PAPI_L2_DCA	PAPI_L2_DCH	PAPI_L2_DCR	PAPI_L2_ICA	PAPI_L2_ICH	PAPI_L2_ICR	PAPI_L2_TCM	PAPI_L2_TCW	PAPI_L3_DCR	PAPI_L3_DCW	PAPI_L3_TCA	PAPI_L3_TCM	PAPI_TLB_DM	PAPI_STL_ICY
2D4PStencil_E			×							×	×		×							×		
2PStencil_E			×							×			×							×		×
Add_E			×							×			×							×		×
Collatz			×	×	×	×			×													
Copy_E					×					×			×					×	×			
Friends	×					×						×	×									×
MatrixMultNOpt								×		×	×				×						×	
MatrixMult	×		×					×			×					×						
Matxvec				×		×				×			×							×		
Reduction_E			×		×					×			×							×		
Rows_E								×			×									×	×	×
Scale_E			×	×						×			×							×		
Stencil	×							×			×		×	×								
Stride16_E										×			×							×	×	×
Stride2_E										×			×				×			×		×
Stride4_E								×		×			×				×					×
Stride64_E				×												×			×	×	×	×
Triad_E		×								×			×					×	×			

where X denotes the number of threads, $T_t(X)$ and $E(X)$ denote the obtained execution time and efficiency of a code-region using this number of threads. Therefore, $P_i(X)$ relates execution time with resource efficiency, allowing to automatically find the number of threads that maximizes performance (minimizing execution time) without wasting resources.

$P_i(X)$ captures the trade-off between execution time and thread utilization. Configurations with long execution times or poor efficiency are penalized, while configurations that achieve fast execution without wasting cores yield a lower index. Minimizing $P_i(X)$ identifies the optimal number of threads that balances runtime and resource usage. For example, if doubling the number of threads only slightly reduces execution time, efficiency decreases and $P_i(X)$ increases, signaling that adding more threads is not beneficial.

5. Evaluation of the Proposed Methodology

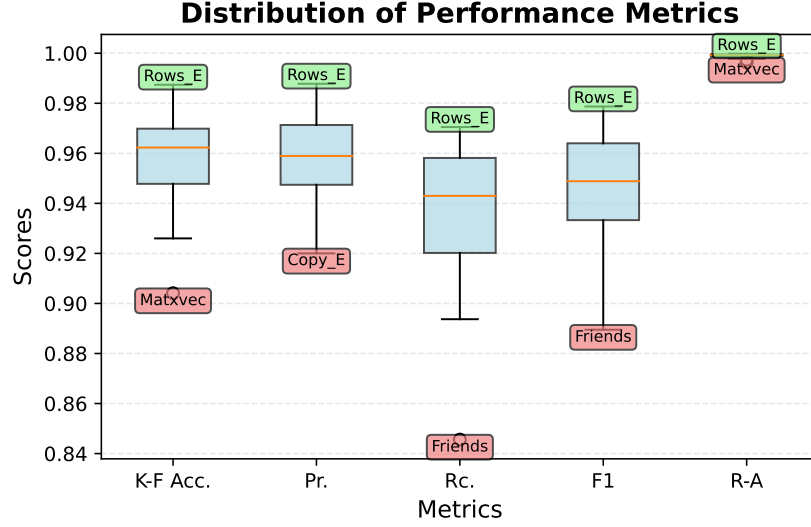


Figure 5.1: Evaluation results of the ensemble for the prediction of the number of threads TP minimizing $P_i(X)$.

Figure 5.1 presents the results of the ensemble methodology for optimizing the number of threads for all code-regions, according to the objective function that minimizes $P_i(X)$. The detailed breakdown of these results is provided in Table 5.10. The results show that the ensemble achieves consistently high classification accuracy scores ranging from 0.9041 to 0.9974, while using only 5-6 HwPCs. The ensemble consistently reaches high precision and recall rates. The high ROC AUC scores (0.9963-0.9998) confirm robust discriminative capability across different code-regions, demonstrating the ensemble’s efficiency and practical applicability. These metrics indicate that the methodology reliably identifies the optimal number of threads’ configurations for the code-regions considered.

The comparative analysis with OpenTuner, which optimizes execution time, a second ensemble model was trained using minimum execution time as the objective function. This alternative model not only provides a basis for a fair comparison, but also reinforces the adaptability of our proposed methodology to different optimization targets. Figure 5.2 shows that the ensemble has consistently strong performance, maintaining high classification accuracy across all evaluation metrics, with K-fold accuracies ranging from 0.7987 to 0.9937. The variation in accuracy

5. Evaluation of the Proposed Methodology

Table 5.10: Full evaluation results of the ensemble for the prediction of the number of threads TP minimizing $P_i(X)$.

Code-region / Metric	K-Fold Accuracy	Precision	Recall	F1-Score	ROC AUC	Reduced
2D4PStencil_E	0.9621	0.9711	0.9584	0.9645	0.9994	5
2PStencil_E	0.9633	0.9711	0.9574	0.9640	0.9996	5
Add_E	0.9475	0.9593	0.9282	0.9428	0.9993	5
Copy_E	0.9260	0.9200	0.9001	0.9092	0.9977	5
MatrixMult	0.9774	0.9546	0.9434	0.9473	0.9993	5
MatrixMultNOpt	0.9774	0.9627	0.9608	0.9611	0.9996	5
Matxvec	0.9041	0.9250	0.8937	0.9072	0.9963	5
Reduction_E	0.9688	0.9762	0.9607	0.9682	0.9996	5
Rows_E	0.9874	0.9878	0.9705	0.9787	0.9998	5
Scale_E	0.9422	0.9459	0.9176	0.9310	0.9988	5
Stencil	0.9487	0.9520	0.9326	0.9402	0.9993	5
Stride16_E	0.9702	0.9717	0.9426	0.9563	0.9989	6
Stride2_E	0.9569	0.9586	0.9438	0.9504	0.9993	5
Stride4_E	0.9312	0.9411	0.9150	0.9269	0.9974	5
Stride64_E	0.9679	0.9714	0.9570	0.9640	0.9980	5
Triad_E	0.9625	0.9720	0.9625	0.9670	0.9995	5
Collatz	0.9726	0.9573	0.9280	0.9419	0.9997	5
Friends	0.9546	0.9427	0.8456	0.8895	0.9992	6

stems from the inherent characteristic of execution time optimization: multiple number of threads configurations often achieve nearly identical execution times close to the minimum, and for certain problem sizes, different number of threads may achieve the optimal performance. This makes perfect classification challenging, as predictions that are slightly off the absolute minimum may still be practically optimal. Nevertheless, the high precision, recall, and F1-scores indicate that the approach effectively identifies optimal number of thread configurations for minimizing execution time with a reduced set of 4-7 HwPCs. This confirms the methodology’s effectiveness to target execution time reduction. The detailed breakdown of these results is provided in Table 5.11.

Figure 5.3 compares the optimal performance index (Equation 5.1) with the one obtained by the ensemble methodology (optimizing $P_i(X)$) and by OpenTuner (optimizing execution time). As expected, the ensemble methodology achieves optimal or almost optimal results for all regions, while OpenTuner consistently produces worse (i.e., higher) performance indices across all code-regions, indicating

5. Evaluation of the Proposed Methodology

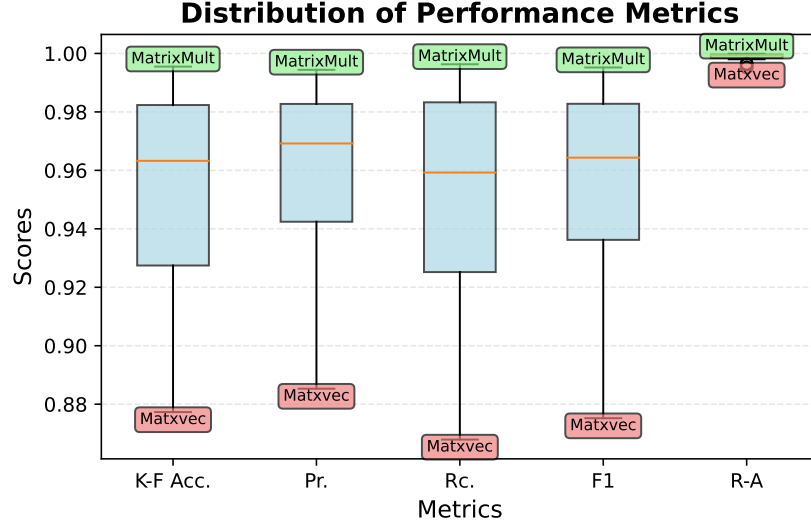


Figure 5.2: Evaluation results of the ensemble for the prediction of the number of threads TP minimizing the execution time.

Table 5.11: Full evaluation results of the ensemble for the prediction of the number of threads TP minimizing the execution time.

Code-region / Metric	K-Fold Accuracy	Precision	Recall	F1-Score	ROC AUC	Reduced
2D4PStencil_E	0.9858	0.9886	0.9855	0.9870	0.9998	5
2PStencil_E	0.9653	0.9719	0.9605	0.9658	0.9994	5
Add_E	0.9378	0.9503	0.9321	0.9401	0.9989	5
Copy_E	0.9159	0.9282	0.9144	0.9203	0.9980	6
MatrixMult	0.9955	0.9944	0.9963	0.9952	0.9999	4
MatrixMultNOpt	0.9865	0.9883	0.9855	0.9867	0.9998	4
Matxvec	0.8773	0.8853	0.8679	0.8752	0.9952	5
Reduction_E	0.9833	0.9829	0.9820	0.9824	0.9997	5
Rows_E	0.9579	0.9647	0.9537	0.9588	0.9991	4
Scale_E	0.9701	0.9700	0.9644	0.9665	0.9993	5
Stencil	0.9379	0.9398	0.9365	0.9378	0.9985	5
Stride16_E	0.9018	0.9092	0.8976	0.9017	0.9962	6
Stride2_E	0.9612	0.9684	0.9580	0.9629	0.9995	4
Stride4_E	0.9240	0.9517	0.9229	0.9357	0.9984	7
Stride64_E	0.8939	0.9090	0.8866	0.8961	0.9965	5
Triad_E	0.9880	0.9908	0.9884	0.9895	0.9999	4
Collatz	0.9795	0.9822	0.9837	0.9829	0.9997	7
Friends	0.9713	0.9753	0.9765	0.9758	0.9995	6

less efficient resource utilization. Performance index degradation is particularly

5. Evaluation of the Proposed Methodology

severe for certain code-regions: **Add_E** ($4.33\times$ worse), **Scale_E** ($4.22\times$ worse), **Matxvec** ($4.20\times$ worse), and **Stride64_E** ($4.05\times$ worse). The detailed breakdown of these results is provided in Table 5.12.

Figure 5.4 presents the speedup comparison between OpenTuner, the ensemble methodology (optimizing $P_i(X)$ and execution time), and the optimal speedup. It shows that the ensemble minimizing execution time (Time) achieves speedups closer to the optimal values in the majority of cases (13 out of 18 code-regions), while the ensemble minimizing $P_i(X)$ (Perf) shows larger deviations from optimal values but it obtains better results than OpenTuner (11 out of 18). The detailed breakdown of these results is provided in Table 5.13.

Figure 5.5 shows the comparison of the time needed to compute the number of threads by the ensemble and OpenTuner. The ensemble models consistently deliver 2-7 seconds when optimizing $P_i(X)$ and 3-7 seconds when optimizing speedup, ensuring low predictable optimization overhead in all scenarios. In contrast, OpenTuner exhibits significant time variability, with execution times ranging from 2 seconds to over 4 minutes. The detailed breakdown of these results is provided in Table 5.14.

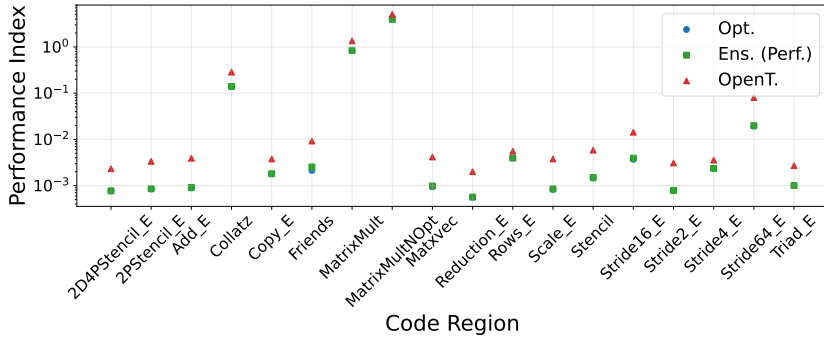


Figure 5.3: Comparison of the optimal (Opt.), ensemble minimizing $P_i(X)$ (Perf.), and OpenTuner (OpenT.) mean performance indices.

This combination of generally superior speedup performance and consistently lower execution times reinforces the ensemble’s practical advantage for performance-critical optimization scenarios.

5. Evaluation of the Proposed Methodology

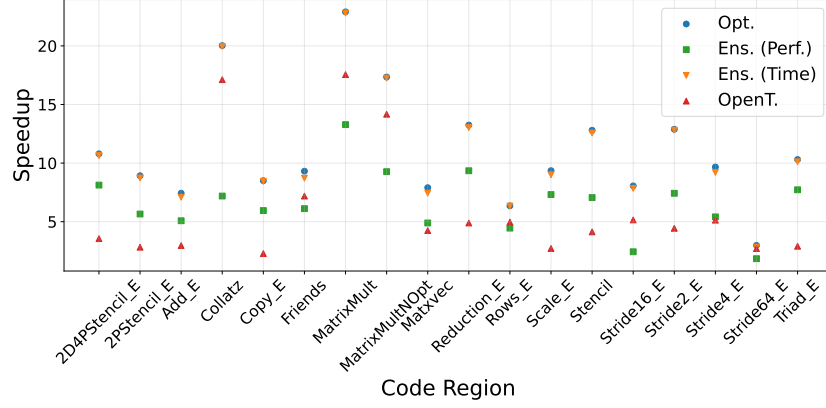


Figure 5.4: Comparison of the optimal (Opt.), ensemble minimizing $P_i(X)$ (Perf.), ensemble minimizing execution time (Time), and OpenTuner (OpenT.) mean speedups.

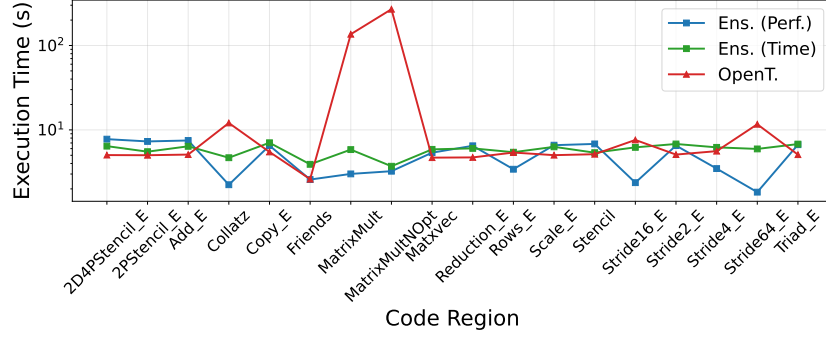


Figure 5.5: Comparison of the execution times (logarithmic scale) required for computing the number of threads: ensemble minimizing $P_i(X)$ (Perf.), ensemble minimizing execution time (Time), and OpenTuner (OpenT.).

5.2.4 Tuning Parameter: Thread Affinity Policy

The next evaluated TP is thread affinity policy. The optimal affinity policy a^* for a given code-region is defined as the policy that minimizes execution time:

$$a^* = \min_{a \in A} T(a) \quad (5.2)$$

where $A = \{\text{close}, \text{spread}\}$ is the set of available affinity policies and $T(a)$ represents the execution time under affinity policy a . The methodology predicts the affinity policy that achieves the minimum execution time. Prediction accuracy is

5. Evaluation of the Proposed Methodology

Table 5.12: Full comparison of the optimal, ensemble minimizing $P_i(X)$ (Perf), and OpenTuner mean performance indices.

Code-region / Performance Index	Optimal	Ensemble	OpenTuner
2D4PStencil_E	0.0008	0.0008	0.0023
2PStencil_E	0.0008	0.0009	0.0034
Add_E	0.0009	0.0009	0.0039
Collatz	0.1372	0.1401	0.2856
Copy_E	0.0018	0.0018	0.0038
Friends	0.0021	0.0025	0.0092
MatrixMult	0.8310	0.8314	1.3557
MatrixMultNOpt	3.9343	3.9437	5.1013
Matxvec	0.0009	0.0010	0.0042
Reduction_E	0.0006	0.0006	0.0020
Rows_E	0.0039	0.0040	0.0056
Scale_E	0.0008	0.0009	0.0038
Stencil	0.0015	0.0015	0.0059
Stride16_E	0.0037	0.0039	0.0143
Stride2_E	0.0008	0.0008	0.0031
Stride4_E	0.0023	0.0024	0.0036
Stride64_E	0.0193	0.0198	0.0801
Triad_E	0.0010	0.0010	0.0027

measured as the percentage of correct identifications of a^* .

Figure 5.6 shows that the ensemble methodology maintains high performance for affinity TP prediction. The ensemble achieves high K-fold accuracies ranging from 0.9512 to 0.9993, with most code-regions exceeding 97% accuracy. The strong precision (0.9474-0.9991) and recall (0.9229-0.9990) values confirm reliable identification of optimal affinity configurations. The methodology accomplishes this performance using only 3-7 HwPCs. These results show the ensemble’s effective feature reduction while maintaining strong discriminative capability for thread affinity optimization. The detailed breakdown of these results is provided in Table 5.15.

Figure 5.7 shows that OpenTuner underperforms across all evaluation metrics for affinity tuning. K-fold accuracies range from 0.3427 to 0.6433, with precision, recall, and F1-scores consistently below 0.6, indicating poor classification performance barely exceeding random selection. Time percentage differences reveal substantial performance degradation, with most benchmarks showing deterioration

5. Evaluation of the Proposed Methodology

Table 5.13: Full comparison of the optimal, ensemble minimizing $P_i(X)$ (Perf.), ensemble minimizing execution time (Time), and OpenTuner mean speedups.

Code-region	Optimal	Ensemble (Performance)	Ensemble (Time)	OpenTuner
2D4PStencil_E	10.79x	8.12x	10.64x	3.56x
2PStencil_E	8.92x	5.65x	8.73x	2.82x
Add_E	7.43x	5.08x	7.08x	2.96x
Collatz	20.02x	7.19x	19.96x	17.12x
Copy_E	8.50x	5.95x	8.44x	2.28x
Friends	9.31x	6.11x	8.71x	7.19x
MatrixMult	22.90x	13.28x	22.80x	17.55x
MatrixMultNOpt	17.34x	9.27x	17.25x	14.17x
Matxvec	7.90x	4.89x	7.43x	4.25x
Reduction_E	13.24x	9.35x	13.03x	4.89x
Rows_E	6.35x	4.45x	6.33x	4.96x
Scale_E	9.35x	7.31x	9.00x	2.72x
Stencil	12.80x	7.06x	12.56x	4.13x
Stride16_E	8.06x	2.43x	7.84x	5.15x
Stride2_E	12.88x	7.43x	12.80x	4.44x
Stride4_E	9.67x	5.41x	9.21x	5.13x
Stride64_E	2.98x	1.85x	2.83x	2.71x
Triad_E	10.31x	7.72x	10.11x	2.90x

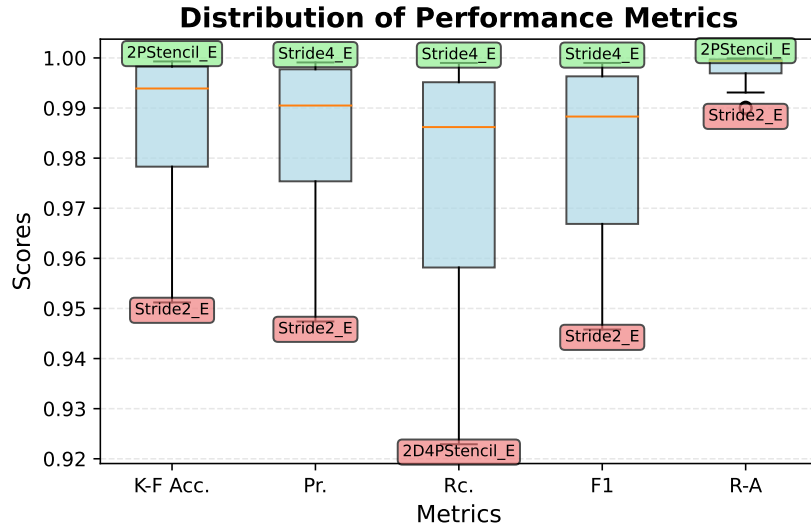


Figure 5.6: Evaluation results of the ensemble for the prediction of the affinity TP.

up to 39.96%. The detailed breakdown of these results is provided in Table 5.16.

Finally, looking at the execution times between the ensemble and OpenTuner

5. Evaluation of the Proposed Methodology

Table 5.14: Full comparison of the execution times required for computing the number of threads: ensemble minimizing $P_i(X)$ (Perf.), ensemble minimizing execution time (Time), and OpenTuner.

Code-region / Execution Time	Ensemble (Performance)	Ensemble (Time)	OpenTuner
2D4PStencil_E	7.77s	6.42s	5.03s
2PStencil_E	7.30s	5.52s	5.01s
Add_E	7.49s	6.39s	5.10s
Collatz	2.24s	4.68s	12.08s
Copy_E	6.52s	7.04s	5.48s
Friends	2.58s	3.91s	2.61s
MatrixMult	3.01s	5.84s	2m 16s
MatrixMultNOpt	3.24s	3.71s	4m 29s
Matxvec	5.37s	5.88s	4.69s
Reduction_E	6.48s	6.04s	4.72s
Rows_E	3.42s	5.44s	5.39s
Scale_E	6.59s	6.30s	5.02s
Stencil	6.82s	5.37s	5.15s
Stride16_E	2.37s	6.21s	7.60s
Stride2_E	6.52s	6.81s	5.10s
Stride4_E	3.48s	6.21s	5.60s
Stride64_E	1.83s	5.95s	11.66s
Triad_E	6.68s	6.79s	5.09s

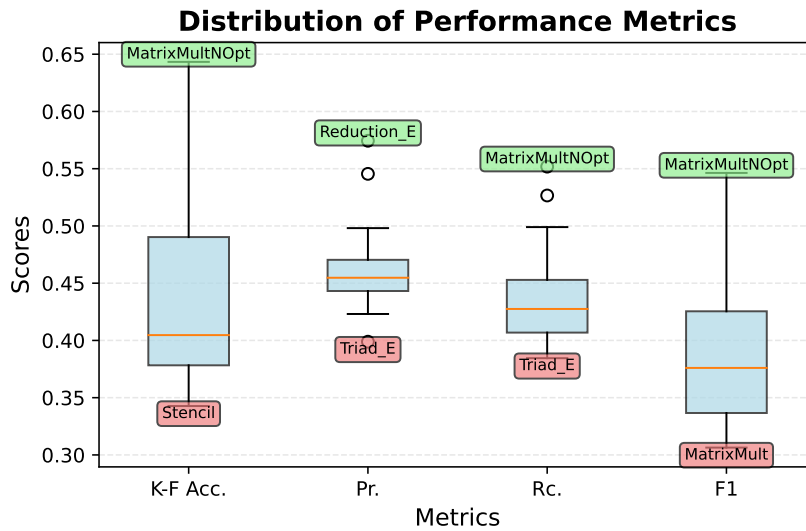


Figure 5.7: Evaluation results of OpenTuner for the prediction of the affinity TP.

5. Evaluation of the Proposed Methodology

Table 5.15: Full evaluation results of the ensemble for the prediction of the affinity TP.

Code-region / Metric	K-Fold Accuracy	Precision	Recall	F1-Score	ROC AUC	Reduced
2D4PStencil_E	0.9756	0.9731	0.9229	0.9460	0.9901	4
2PStencil_E	0.9993	0.9989	0.9974	0.9982	0.9999	4
Add_E	0.9979	0.9965	0.9947	0.9956	0.9997	4
Copy_E	0.9984	0.9979	0.9936	0.9958	0.9997	5
MatrixMult	0.9967	0.9972	0.9957	0.9965	0.9994	7
MatrixMultNOpt	0.9987	0.9986	0.9987	0.9987	0.9999	4
Matxvec	0.9623	0.9585	0.9464	0.9522	0.9931	5
Reduction_E	0.9943	0.9914	0.9848	0.9880	0.9995	5
Rows_E	0.9869	0.9859	0.9781	0.9820	0.9983	4
Scale_E	0.9792	0.9677	0.9436	0.9552	0.9957	4
Stencil	0.9780	0.9779	0.9767	0.9773	0.9976	5
Stride16_E	0.9935	0.9927	0.9922	0.9924	0.9994	4
Stride2_E	0.9512	0.9474	0.9442	0.9458	0.9899	4
Stride4_E	0.9992	0.9991	0.9990	0.9990	0.9999	3
Stride64_E	0.9946	0.9896	0.9876	0.9886	0.9995	4
Triad_E	0.9984	0.9979	0.9953	0.9966	0.9999	4
Collatz	0.9792	0.9747	0.9769	0.9758	0.9978	6
Friends	0.9742	0.9774	0.9520	0.9639	0.9967	6

(Figure 5.8), the ensemble shows a higher efficiency with consistently low execution times (0.42-0.59s) across all code-regions. In contrast, OpenTuner exhibits highly variable performance, ranging from 0.79 seconds to under 1.5 minutes, with poor scalability for compute-intensive code-regions like **MatrixMult** (32.91s) and **MatrixMultNOpt** (1m 33s). Overall, the ensemble methodology achieves from 2 to 180× faster optimization times while maintaining a higher classification performance reinforcing its practical advantage for runtime tuning of affinity TP. The detailed breakdown of these results is provided in Table 5.16.

5.2.5 Tuning Parameter: Scheduling Policy and Chunk Size

The final TPs we consider are the scheduling policy and the chunk size. The optimal scheduling policy s^* and corresponding chunk size c^* are determined by:

$$(s^*, c^*) = \min_{(s, c) \in S \times C} T(s, c) \quad (5.3)$$

5. Evaluation of the Proposed Methodology

Table 5.16: Full evaluation results of OpenTuner for the prediction of the affinity TP.

Code-region / Metric	K-Fold Accuracy	Precision	Recall	F1-Score
2D4PStencil_E	0.3772	0.4546	0.4091	0.3348
2PStencil_E	0.3815	0.4705	0.4247	0.3279
Add_E	0.3513	0.4582	0.4203	0.3216
Copy_E	0.6369	0.4981	0.4990	0.4770
MatrixMult	0.3513	0.4609	0.4032	0.3064
MatrixMultNOpt	0.6433	0.5455	0.5518	0.5463
Matxvec	0.5259	0.4415	0.4568	0.4374
Reduction_E	0.5679	0.5744	0.5266	0.4559
Rows_E	0.4138	0.4482	0.4368	0.4013
Scale_E	0.4009	0.4498	0.3966	0.3445
Stencil	0.3427	0.4881	0.4891	0.3421
Stride16_E	0.3664	0.4525	0.4061	0.3279
Stride2_E	0.4084	0.4267	0.4303	0.4067
Stride4_E	0.5075	0.4326	0.4310	0.4317
Stride64_E	0.3901	0.4231	0.4198	0.3893
Triad_E	0.4289	0.3989	0.3845	0.3865
Collatz	0.4386	0.4549	0.4053	0.3655
Friends	0.3815	0.4701	0.4411	0.3446

where $S = \{\text{static}, \text{dynamic}, \text{guided}\}$ is the set of OpenMP scheduling policies and C is the discrete set of evaluated chunk sizes. The methodology predicts the policy-size combination that minimizes execution time. These TPs are evaluated through classification accuracy (correctly predicting the configuration with minimum execution time) since they represent discrete categorical choices rather than continuous resource allocation.

Table 5.18 presents evaluation results for scheduling policy and chunk size TPs optimization on two code-regions. The ensemble methodology demonstrates high performance for both TPs, achieving K-fold accuracies exceeding 98% across both cases. The results are consistently high across the rest of the presented metrics for both TPs. This shows that the ensemble can effectively identify optimal configurations for both TPs using only 5-6 HwPCs, demonstrating its versatility in handling diverse TPs beyond number of threads or thread affinity.

Table 5.19 summarizes the results that show that OpenTuner performs poorly for both scheduling policy and chunk size TP optimization. K-fold accuracies

5. Evaluation of the Proposed Methodology

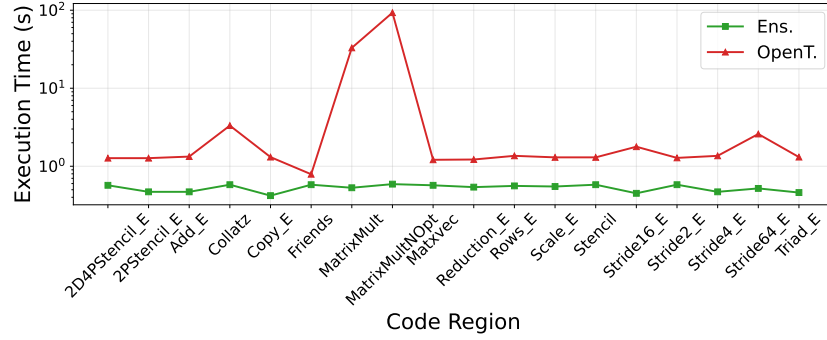


Figure 5.8: Comparison of the execution times (logarithmic scale) required for computing affinity: ensemble and OpenTuner.

Table 5.17: Full comparison of the execution times (logarithmic scale) required for computing affinity: ensemble and OpenTuner.

Code-region / Execution Time	Ensemble	OpenTuner
2D4PStencil_E	0.57s	1.27s
2PStencil_E	0.47s	1.27s
Add_E	0.47s	1.33s
Collatz	0.58s	3.32s
Copy_E	0.42s	1.31s
Friends	0.58s	0.79s
MatrixMult	0.53s	32.91s
MatrixMultNOpt	0.59s	1m 33s
Matxvec	0.57s	1.21s
Reduction_E	0.54s	1.22s
Rows_E	0.56s	1.36s
Scale_E	0.55s	1.3s
Stencil	0.58s	1.3s
Stride16_E	0.45s	1.78s
Stride2_E	0.58s	1.28s
Stride4_E	0.47s	1.36s
Stride64_E	0.52s	2.59s
Triad_E	0.46s	1.31s

range from 0.3367 to 0.4224, indicating performance barely above random selection. Precision, recall, and F1-scores consistently remain below 0.4 across all cases, demonstrating inadequate classification capability for both TPs. Additionally, OpenTuner shows performance degradation with time percentage differences of 2.32-4.67%. Contrasting these results with the ensemble methodology’s accuracy >98%, further confirms the ensemble’s capabilities for TP optimization.

5. Evaluation of the Proposed Methodology

Table 5.18: Scheduling policy and chunk size parameter evaluation results for the expanded benchmark on the Xeon E5-4620.

Tuning Parameter	Code-region / Metric	K-Fold Accuracy	Precision	Recall	F1-Score	ROC AUC	Reduced
Scheduling Policy	Collatz	0.9883	0.9848	0.9831	0.9839	0.9992	5
	Friends	0.9893	0.9924	0.9917	0.9921	0.9994	6
Chunk Size	Collatz	0.9897	0.9913	0.9799	0.9853	0.9995	5
	Friends	0.9853	0.9824	0.9820	0.9821	0.9994	5

Finally, considering the execution times between the ensemble methodology and OpenTuner for scheduling and chunk size optimization (Table 5.20) the results show no clear time-wise advantage between them, as each method outperforms the other on different benchmarks. These divergent results highlight the complexity of scheduling and chunk size optimization.

Table 5.19: Scheduling policy and chunk size parameter OpenTuner evaluation results for the expanded benchmark on the Xeon E5-4620.

Tuning Parameter	Code-region / Metric	K-Fold Accuracy	Precision	Recall	F1-Score
Scheduling Policy	Collatz	0.4224	0.3702	0.3656	0.3654
	Friends	0.3637	0.3328	0.2813	0.2866
Chunk Size	Collatz	0.4165	0.3856	0.3921	0.3785
	Friends	0.3367	0.3556	0.3509	0.3341

Table 5.20: Comparison of the ensemble, and OpenTuner mean execution times for the expanded benchmark on the Xeon E5-4620. Focus on scheduling policy and chunk size.

Code-region / Execution Time	Ensemble	OpenTuner
Collatz	1.34s	4.18s
Friends	1.61s	0.78s

5.3 Application to the NAS Parallel Benchmarks

5.3.1 Evaluation Dataset

To ensure a comprehensive and representative assessment, 7 OpenMP code-regions from NAS Parallel Benchmarks (NPB) [8] were extracted (summarized in Table

5. Evaluation of the Proposed Methodology

5.21). The code-regions represent a diverse range of computational patterns. The NPB suite is widely recognized and extensively used in parallel computing to evaluate the performance of parallel systems. It's important to mention that this dataset was not used for training, it was only used for evaluation purposes. As previously, PAPI was used to gather the HwPC data.

- **BT benchmark:** Extracted code-region - `add_BT`. This code-region performs element-wise addition of two 4D arrays (u and rhs), updating the solution vector in the Block Tri-diagonal solver. The computation involves four nested loops iterating over grid points with five components per point, representing a regular, memory-bound operation.
- **CG benchmark:** Extracted code-regions representing key operations in the Conjugate Gradient iterative solver:
 1. `normztox_CG`: Vector scaling operation that normalizes vector z and stores the result in x .
 2. `norm_temps_CG`: Computes intermediate norm values using reduction operations on vectors x and z .
 3. `rhorr_CG`: Calculates the dot product of vector r with itself using parallel reduction.
 4. `pr_beta_p_CG`: Updates the search direction vector p by combining r and scaled p .
 5. `qAp_CG`: Sparse matrix-vector multiplication computing $q = A \times p$, utilizing compressed row storage format.
- **LU benchmark:** Extracted code-region - `l2norm_LU`. This code-region computes the L2 norm of a 3D vector field with five components. It uses a combination of parallel loops and critical sections to accumulate partial sums, followed by normalization. The computation involves reduction operations across three spatial dimensions.

5. Evaluation of the Proposed Methodology

Table 5.21: NAS Parallel Benchmark code-regions grouped by benchmark suite.

Benchmark	Code-region	Description
Block Tri-diagonal Solver - BT (1 code-region)	add_BT	Element-wise addition of 4D arrays updating solution vector across grid points
Conjugate Gradient - CG (5 code-regions)	normztox_CG	Vector scaling operation normalizing vector z to x
	norm_temps_CG	Computes intermediate norm values with parallel reduction
	rhorr_CG	Dot product calculation using parallel reduction
	pr_beta_p_CG	Search direction vector update combining r and scaled p
	qAp_CG	Sparse matrix-vector multiplication in compressed row storage
Lower-Upper Gauss-Seidel Solver - LU (1 code-region)	l2norm_LU	L2 norm computation of 3D vector field with five components

5.3.2 Code-Region Identification

To validate the proposed methodology, we conducted comprehensive experiments using the NPB dataset. The initial evaluation focuses on code-region identification by leveraging an ensemble of models trained on the comprehensive dataset from Section 5.2.

Figure 5.9 presents code-region identification results using the already trained ensemble that contains a minimal set of HwPCs. The ensemble demonstrates strong code-region identification performance across most code-regions, achieving high accuracy scores. The high F1 scores (>0.96) suggests the ensemble is highly confident of the predictions. The detailed breakdown of these results is provided in Table 5.22

5.3.3 Tuning Parameter: Number of Optimal Threads

The evaluation continues with the optimization of the number of threads, focusing on efficiency as defined by the performance index (Equation 5.1). Figure 5.10 compares the optimal performance index (Equation 5.1) with the one obtained by the ensemble methodology (optimizing $P_i(X)$) and by OpenTuner (optimizing

5. Evaluation of the Proposed Methodology

Table 5.22: Full evaluation results of the ensemble for the code-region identification (NAS benchmarks).

Code-region / Metric	Expected Code-region	K-Fold Accuracy	Precision	Recall	F1-Score
Add_BT	Triad_E	0.9650	0.9999	0.9650	0.9822
l2norm_LU	Reduction_E	0.8219	0.9999	0.8219	0.9022
norm_temps_CG	Reduction_E	0.9760	0.9999	0.9760	0.9878
normztox_CG	Triad_E	0.9583	0.9999	0.9583	0.9787
pr_beta_p_CG	Triad_E	0.9719	0.9999	0.9719	0.9858
qAp_CG	Reduction_E	0.9308	0.9999	0.9308	0.9642
rhorr_CG	Reduction_E	0.9625	0.9999	0.9625	0.9809

execution time). The results obtained by the ensemble are usually close to the optimal value, and, in all cases, similar to the ones obtained by OpenTuner. These results hint that, for these regions, the difference between the number of threads minimizing $P_i(X)$ and the one minimizing execution time is not significant. The detailed breakdown of these results is provided in Table 5.23.

Figure 5.11 presents the speedup comparison between OpenTuner, the ensemble methodology (optimizing $P_i(X)$ and execution time), and the optimal speedup. It shows that, in effect, the execution time difference optimizing $P_i(X)$ and op-

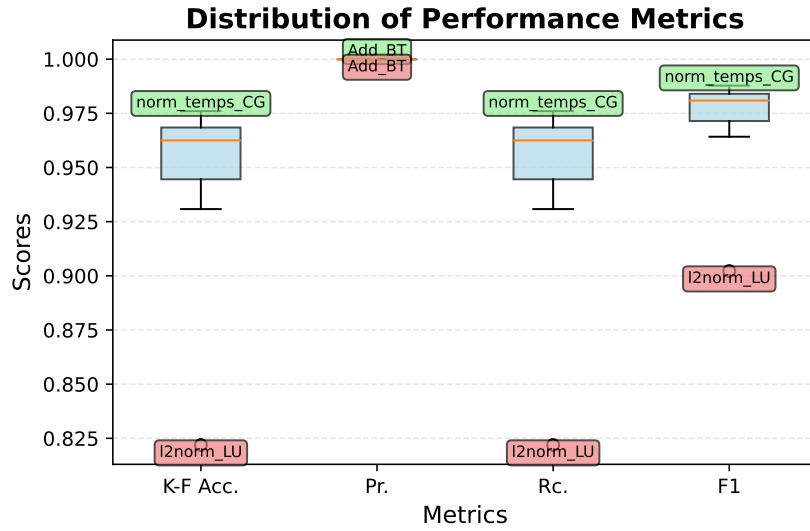


Figure 5.9: Evaluation results of the ensemble for the code-region identification (NAS benchmarks).

5. Evaluation of the Proposed Methodology

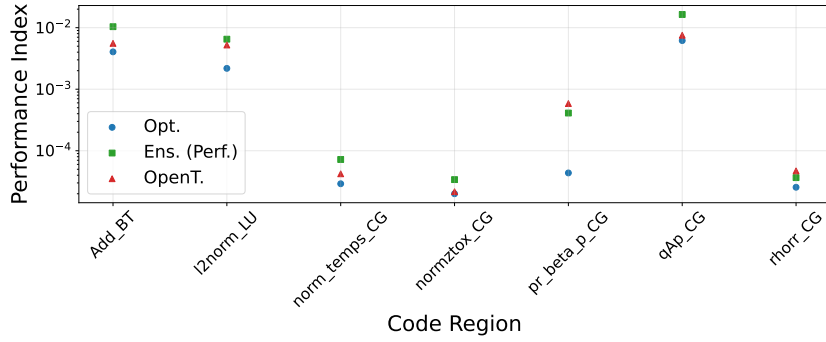


Figure 5.10: Comparison of the optimal (Opt.), ensemble minimizing $P_i(X)$ (Perf), and OpenTuner (OpenT.) mean performance indices (NAS benchmarks).

timizing time are not significant in most cases (except for the qAp_CG case). Consequently, the values obtained by the ensemble methodology and OpenTuner are quite close. The detailed breakdown of these results is provided in Table 5.24.

Figure 5.12 shows the comparison of the time needed to compute the number of threads by the ensemble and OpenTuner. The ensemble models consistently completes optimization in under a minute for all cases: 2-44 seconds when minimizing $P_i(X)$ and 2-50 seconds when maximizing speedup, ensuring low predictable optimization overhead in all scenarios. In contrast, OpenTuner exhibits significant time variability, with execution times ranging from approximately 18 minutes for CG regions to nearly 3 hours for BT regions. This means that the ensemble methodology is 20 to 300x faster than OpenTuner in these cases. The detailed breakdown of these results is provided in Table 5.25.

Table 5.23: Full comparison of the optimal, ensemble minimizing $P_i(X)$ (Perf), and OpenTuner mean performance indices (NAS benchmarks).

Code-region / Performance Index	Optimal	Ensemble	OpenTuner
Add_BT	0.00405	0.01038	0.00556
l2norm_LU	0.00218	0.00650	0.00523
norm_temps_CG	0.00003	0.00007	0.00004
normztox_CG	0.00002	0.00003	0.00002
pr_beta_p_CG	0.00004	0.00041	0.00059
qAp_CG	0.00617	0.01638	0.00754
rhorr_CG	0.00003	0.00004	0.00005

5. Evaluation of the Proposed Methodology

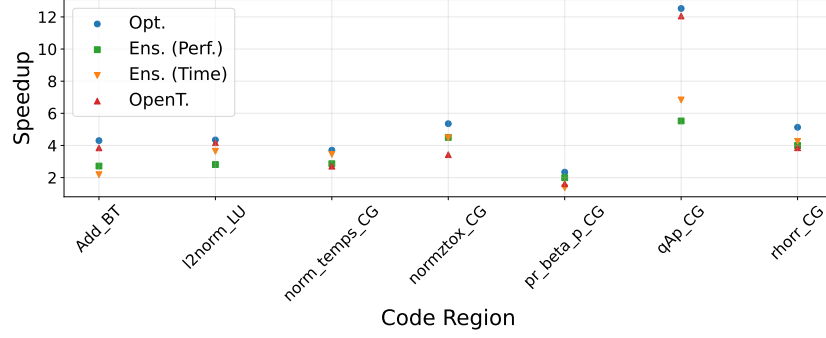


Figure 5.11: Comparison of the optimal (Opt.), ensemble minimizing $P_i(X)$ (Perf), ensemble minimizing execution time (Time), and OpenTuner (OpenT.) mean speedups (NAS benchmarks).

Table 5.24: Full comparison of the optimal, ensemble minimizing $P_i(X)$ (Perf), ensemble minimizing execution time (Time), and OpenTuner mean speedups (NAS benchmarks).

Code-region / Speedup Factor	Optimal	Ensemble (Performance)	Ensemble (Time)	OpenTuner
Add_BT	4.30x	2.72x	2.18x	3.85x
l2norm_LU	4.35x	2.81x	3.64x	4.17x
norm_temps_CG	3.71x	2.86x	3.45x	2.71x
normztox_CG	5.35x	4.50x	4.48x	3.43x
pr_beta_p_CG	2.34x	1.99x	1.36x	1.61x
qAp_CG	12.52x	5.53x	6.83x	12.06x
rhorr_CG	5.13x	4.00x	4.25x	3.86x

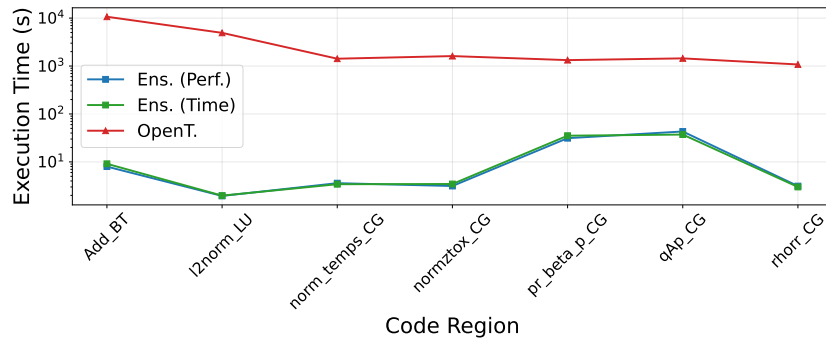


Figure 5.12: Comparison of the execution times (logarithmic scale) required for computing the number of threads: ensemble minimizing $P_i(X)$ (Perf), ensemble minimizing execution time (Time), and OpenTuner (OpenT) (NAS benchmarks).

5. Evaluation of the Proposed Methodology

Table 5.25: Full comparison of the execution times required for computing the number of threads: ensemble minimizing $P_i(X)$ (Perf), ensemble minimizing execution time (Time), and OpenTuner (NAS benchmarks).

Code-region / Execution Time	Ensemble (Performance)	Ensemble (Time)	OpenTuner Time
Add_BT	7.98s	9.13s	2h 57m 52s
l2norm_LU	1.95s	1.98s	1h 22m 05s
norm_temps_CG	3.6s	3.42s	23m 42s
normztox_CG	3.14s	3.46s	26m 53s
pr_beta_p_CG	31.24s	35.2s	22m 09s
qAp_CG	43.11s	37.18s	24m 05s
rhorr_CG	3.13s	3.0s	17m 59s

5.3.4 Tuning Parameter: Thread Affinity Policy

Thread affinity policy is the next TP evaluated. After identifying the code-region, a specialized set of HwPCs is used to predict the thread affinity policy. Figure 5.13 shows varied performances across NAS benchmark code-regions when using the ensemble methodology to predict optimal thread affinity. Several CG code-regions exhibit high accuracy, e.g.: `normztox_CG` (0.9988) or `rhorr_CG` (0.9916), suggesting the ensemble methodology effectiveness for these sparse matrix operation patterns. However, precision, recall, and F1-scores for these code-regions remained approximately 0.5, indicating potential imbalance in prediction classes. The detailed breakdown of these results is provided in Table 5.26

Table 5.26: Full evaluation results of the ensemble for the prediction of the affinity TP (NAS benchmarks).

Code-region / Metric	K-Fold Accuracy	Precision	Recall	F1-Score
Add_BT	0.9117	0.5000	0.4559	0.4769
l2norm_LU	0.7927	0.6595	0.6204	0.6333
norm_temps_CG	0.9204	0.6421	0.5191	0.5190
normztox_CG	0.9988	0.5000	0.4994	0.4997
pr_beta_p_CG	0.9839	0.5000	0.4920	0.4959
qAp_CG	0.9461	0.5000	0.4731	0.4862
rhorr_CG	0.9916	0.5000	0.4958	0.4979

Looking at the OpenTuner’s affinity optimization results (Figure 5.14) even more severe performance issues appear compared to the ensemble. The accuracy values

5. Evaluation of the Proposed Methodology

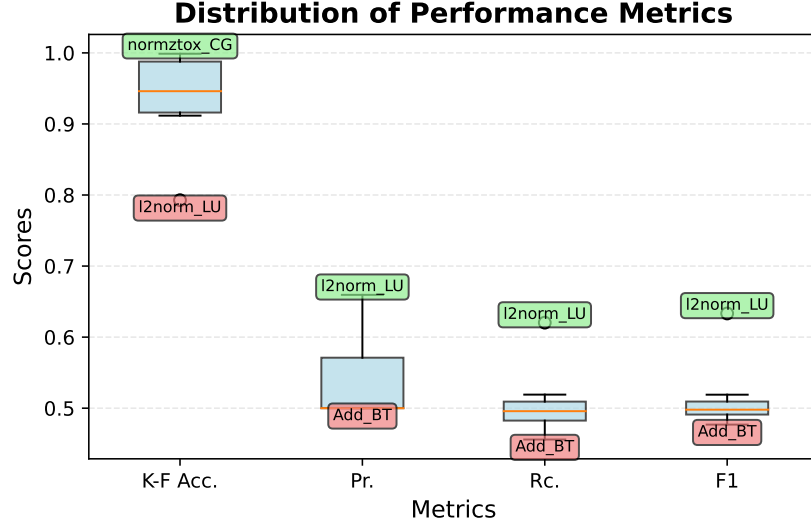


Figure 5.13: Evaluation results of the ensemble for the prediction of the affinity TP (NAS benchmarks).

are consistently poor across all code-regions. As before, OpenTuner suffers from the same class imbalance symptoms with multiple code-regions having low precision and recall values. The detailed breakdown of these results is provided in Table 5.27

Table 5.27: Full evaluation results of OpenTuner for the prediction of the affinity TP (NAS benchmarks).

Code-region / Metric	K-Fold Accuracy	Precision	Recall	F1-Score
Add_BT	0.4375	0.5000	0.2188	0.3043
l2norm_LU	0.5938	0.5389	0.5586	0.5192
norm_temps_CG	0.5375	0.4923	0.4883	0.4618
normztox_CG	0.2625	0.5000	0.1313	0.2079
pr_beta_p_CG	0.2813	0.5000	0.1406	0.2195
qAp_CG	0.2563	0.5000	0.1281	0.2040
rhorr_CG	0.1313	0.5000	0.0656	0.1160

Finally, the execution times for optimal thread affinity prediction between the ensemble methodology and OpenTuner are compared in Figure 5.15. The ensemble shows better execution efficiency for affinity optimization. While the ensemble consistently completes optimization in 1-3 seconds across all code-regions, OpenTuner

5. Evaluation of the Proposed Methodology

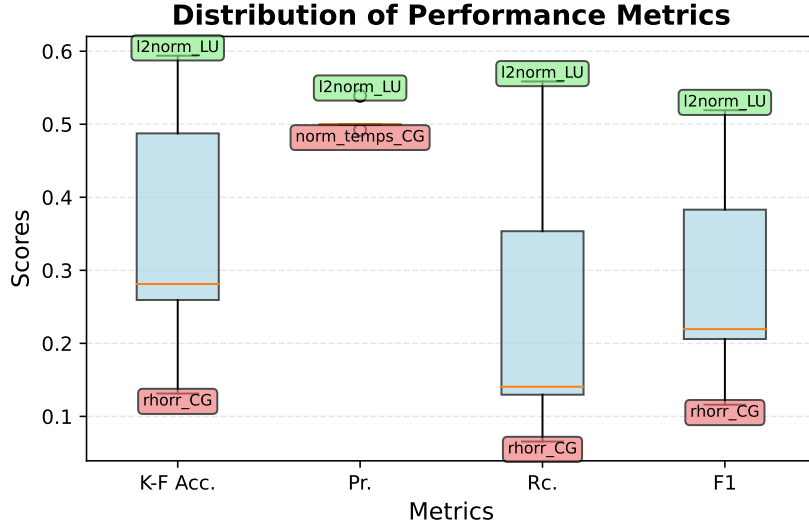


Figure 5.14: Evaluation results of OpenTuner for the prediction of the affinity TP (NAS benchmarks).

requires dramatically longer times ranging from ~ 2 -4 minutes for CG code-regions up to nearly 38 minutes for BT code-regions. Overall, despite both approaches struggle with prediction accuracy due to class imbalance, the ensemble achieves comparable or better performance metrics while being orders of magnitude faster than OpenTuner's long optimization times. The detailed breakdown of these results is provided in Table 5.28

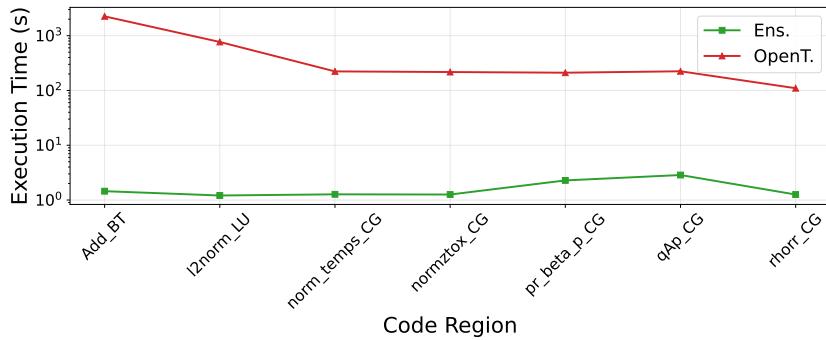


Figure 5.15: Comparison of the execution times (logarithmic scale) required for computing affinity: ensemble and OpenTuner (NAS benchmarks).

5. Evaluation of the Proposed Methodology

Table 5.28: Full comparison of the execution times required for computing affinity: ensemble and OpenTuner (NAS benchmarks).

Code-region / Execution Time	Ensemble	OpenTuner
Add_BT	1.45s	37m 36s
l2norm_LU	1.21s	12m 48s
norm_temps_CG	1.27s	3m 43s
normztox_CG	1.26s	3m 37s
pr_beta_p_CG	2.28s	3m 31s
qAp_CG	2.86s	3m 44s
rhorr_CG	1.26s	1m 50s

5.4 Applying the Methodology to GPUs

The final experiments demonstrate the methodology’s applicability to GPUs by reducing necessary HwPCs for code-region identification and TP optimization.

5.4.1 Evaluation Dataset

HwPC data were collected using the Kernel Tuning Toolkit (KTT) [56] on GeForce GTX 680 (Kepler), GeForce GTX 750 (Maxwell), GeForce GTX 1070 (Pascal), GeForce RTX 2080 (Turing), GeForce RTX 3090 (Ampere), and GeForce RTX 4080 (Ada Lovelace) GPUs across five kernels analyzed by Petrovič et al. [57]: **Convolution**, **Coulomb Sum**, **N-body**, **Transposition**, and **GEMM**. Additional kernels were added to some of the architectures. The **Reduction** kernel was added for Pascal, the **Biconjugate Gradient** and **Hotspot** kernels were added for Turing, and **Biconjugate Gradient** was added for Ada Lovelace. All GPU kernels use CUDA as the parallelization model. Remind, that for GPU applications individual CUDA kernels correspond to code-regions. Table 5.29 provides a summary of the kernels used during the experimentation.

A summary of the Tuning Parameters targeted per architecture is presented in Table 5.30, where **work-group size** balances parallelism against resource consumption, **work-item coarsening** adjusts the computational workload per thread, **local memory caching** enables explicit use of fast shared memory as a cache, **private memory caching** optimizes register usage for fastest data access, **tile**

5. Evaluation of the Proposed Methodology

Table 5.29: GPU kernels grouped by tested architecture.

Architecture	Kernel	Description
Kepler, Maxwell, Pascal, Turing, Ampere, Ada Lovelace	Convolution	2D image convolution with three-pixel halos (input: 2D image, output: convolved 2D image)
	Coulomb Sum	Coulomb summation precomputing atomic charges on a 3D grid (input: set of atoms, output: 3D grid)
	N-body	Gravitational force computation between n bodies for each pair (input: n bodies with positions/masses, output: forces on each body)
	Transposition	Out-of-place matrix transposition (input: matrix, output: transposed matrix)
	GEMM	Matrix multiplication $C = A \times B$ where A is $K \times M$, B is $K \times N$, and C is $N \times M$
Pascal	Reduction	Vector summation (input: vector, output: scalar)
Turing, Ada Lovelace	BiCG	Biconjugate Gradient iterative linear solver (input: matrix and vector, output: solution vector)
Turing	Hotspot	Thermal simulation computing temperature distribution (input: power and initial temperature grid, output: final temperature grid)

size defines memory blocking dimensions to improve locality, **loop unrolling** reduces branching overhead and increases instruction-level parallelism, **local memory padding** prevents bank conflicts in GPU shared memory, and explicit **vectorization** facilitates generation of efficient vector memory instructions.

Table 5.30: Common TPs by kernel. Tile size is ticked when it can be configured differently than Work-Group size.

Kernel / TP	Work-Group Size	Work-item Coarsening	Local Memory Caching	Private Memory Caching	Tile Size	Loop Unrolling	Local Memory Padding	Vectorization
Conv.	×	×	×	×	×	×	×	×
Coul. Sum	×	×				×		×
N-body	×	×	×	×		×		×
Transp.	×	×	×		×		×	×
GEMM	×	×	×		×	×		×
Reduc.	×	×						×
BiCG	×	×	×		×	×		
Hotspot	×	×	×		×	×		

5.4.2 Tuning Parameter Evaluation Results

The ensemble methodology was able to greatly reduce the number of HwPCs for kernel identification.

The resulting reduced sets, shown in Table 5.31, comprise HwPCs of two types as defined by Filipovič et al. in [23]. GPU HwPCs are categorized into two types: $HwPC_{Stress}$, which measure the stress (utilization) of a processor component as a percentage of its peak capacity, and $HwPC_{Ops}$, which count the number of operations performed on a component. For example, for global memory, the percentage of peak bandwidth utilization belongs to $HwPC_{Stress}$, whereas the number of global memory transactions belongs to $HwPC_{Ops}$.

It is worth noting that NVIDIA introduced a new HwPC naming convention with the Volta (Turing) architecture, renaming existing HwPCs and exposing additional profiling granularity. This transition also brought value range rescaling for certain HwPCs (e.g., pre-Volta HwPCs report values in $\langle 0, 10 \rangle$, whereas Volta+ HwPCs report percentages in $\langle 0, 100 \rangle$). Since the equivalence between old and new HwPCs is not completely documented by NVIDIA, the mapping used in this work follows the understanding established by Filipovič et al. [23]. Throughout this section, where possible, these unified abbreviations and categories are used to refer to HwPCs regardless of the architecture generation, allowing for a consistent cross-architecture comparison.

Notably, all selected HwPCs across all architectures belong to the *Ops.* type, with the sole exceptions of `DRAM_U` (Ampere and Ada Lovelace), `flop_sp_efficiency` (Maxwell), `gld_requested_throughput` (Pascal), and `gst_requested_throughput` (Pascal), suggesting that operation counts are more discriminative for kernel identification than utilization metrics. Despite each architecture requiring a distinct reduced set, several patterns emerge across GPU generations. `INST_F32`, which counts executed FP32 instructions, is the most consistent HwPC, appearing in five out of six architectures (all except Pascal), reflecting the central role of floating-point computation in GPU kernel characterization. Similarly, `DRAM_RT`, measuring DRAM read transactions, is selected for four architectures (Maxwell, Turing, Ampere, and Ada Lovelace), highlighting that DRAM read activity is a persistent

5. Evaluation of the Proposed Methodology

Table 5.31: Reduced HwPCs for code-region identification across GPU architectures.

Type	HwPC (Abbr.)	Cat.	GTX 680 (Kepler)	GTX 750 (Maxwell)	GTX 1070 (Pascal)	RTX 2080 (Turing)	RTX 3090 (Ampere)	RTX 4080 (Ada Lovelace)
Inst.	INST_EXE	Ops.	×			×	×	
	INST_LDST	Ops.				×		
	INST_INT	Ops.				×		
	INST_MISC	Ops.				×		
	INST_F32	Ops.	×	×		×	×	×
	inst_per_warp*	Ops.			×			
FP32 Ops.	flop_sp_efficiency*	Stress		×				
DRAM	DRAM_RT	Ops.		×		×	×	×
	DRAM_WT	Ops.		×		×		×
	DRAM_U	Stress					×	×
	dram_read_throughput*	Ops.			×			
L2 Cache	L2_RT	Ops.				×	×	
	L2_WT	Ops.	×	×				
Global Memory	gld_requested_throughput*	Stress			×			
	gst_requested_throughput*	Stress			×			

* HwPC name not in equivalence Table [23]; original name retained.

discriminating feature across generations. As GPU architectures evolve from Kepler to Ada Lovelace, a clear progression in HwPC diversity is observed: older architectures (Kepler and Maxwell) rely on a compact set of instruction and L2 cache HwPCs (INST_F32, INST_EXE, L2_WT), while newer architectures (Turing onwards) require a richer set incorporating fine-grained instruction mix HwPCs (INST_LDST, INST_INT, INST_MISC), reflecting the increased profiling granularity introduced with the Volta architecture. Pascal stands out as the most distinct architecture, relying exclusively on throughput-based HwPCs not present in the equivalence Table [23] (dram_read_throughput, gld_requested_throughput, gst_requested_throughput, inst_per_warp), which are unique to NVIDIA’s pre-Volta profiling API.

As summarized in Figure 5.16 stratified 5-fold cross-validation achieved average accuracy scores of 0.9999 for the four datasets, with near-perfect results across

5. Evaluation of the Proposed Methodology

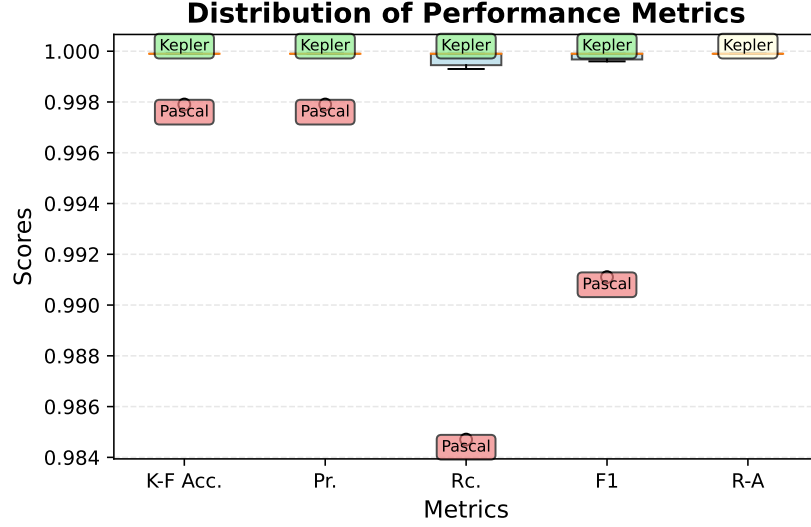


Figure 5.16: Evaluation results of the ensemble for the HwPC reduction and kernel identification for GPUs.

precision, recall, F1-Score, and ROC AUC metrics. This exceptional accuracy suggests highly effective prediction despite significantly different available HwPCs between architectures (32 vs. 167), although the small dataset size and limited number of kernels may contribute to these results. The detailed breakdown of these results is provided in Table 5.32.

Table 5.32: HwPC reduction and kernel identification evaluation results for GPUs.

Architecture / Metric	K-Fold Accuracy	Precision	Recall	F1-Score	ROC AUC	Full Set	Reduced
Kepler	0.9999	0.9999	0.9999	0.9999	0.9999	35	3
Maxwell	0.9999	0.9999	0.9999	0.9999	0.9999	41	5
Pascal	0.9979	0.9979	0.9847	0.9911	0.9999	167	4
Turing	0.9999	0.9999	0.9993	0.9996	0.9999	38	8
Ampere	0.9999	0.9999	0.9999	0.9999	0.9999	37	5
Ada Lovelace	0.9999	0.9999	0.9999	0.9999	0.9999	37	4

Finally, Table 5.33 summarizes kernel-specific TP optimization across the six GPU architectures. In this case we are presenting averaged prediction metrics for different kernels, since each contains varying numbers of TPs. For each kernel we also present the range of HwPCs required for the optimization of the TPs.

The results demonstrate consistently strong performance across all architectures,

5. Evaluation of the Proposed Methodology

with most kernels achieving high accuracy (> 0.81), precision (> 0.81), and recall (> 0.81) values.

Kepler (Figure 5.17), the oldest architecture tested, maintains reliable performance across all kernels, with **N-body** achieving the highest accuracy (0.8593) and **Coulomb Sum** presenting the most challenging case (0.7923 accuracy).

Maxwell (Figure 5.18) shows improved performance over Kepler, particularly notable in **N-body** (0.8972 accuracy), demonstrating the methodology’s adaptability to architectural evolution.

Pascal (Figure 5.19) generally exhibits the strongest performance, particularly evident in the **Reduction** kernel which achieves near-perfect metrics (0.9933 accuracy, F1 score of 0.9937).

Turing (Figure 5.20) shows good performance with notable strength in **N-body** (0.9200 accuracy) and **BiCG** (0.9307 accuracy) kernels.

Ampere (Figure 5.21) demonstrates solid performance across most kernels, although with slightly lower accuracy in **Convolution** and **Coulomb Sum** compared to other architectures.

Ada Lovelace (Figure 5.22) shows improved performance in **Coulomb Sum** (0.8775 accuracy) compared to Ampere, while maintaining fair results for other kernels.

Across all architectures, the **Coulomb Sum** kernel consistently presents the most challenging optimization case, due to the extremely small sample size (209-270 samples). Meanwhile, **N-body** and specialized kernels like **Reduction** and **BiCG** achieve the highest prediction accuracy.

Notably, the ensemble requires only a modest number of HwPCs to achieve these predictions, with most kernels needing between 2-11 HwPCs, indicating efficient feature utilization for accurate TP optimization across different GPU generations. These results validate that the ensemble methodology maintains robust optimization capabilities across diverse GPU architectures.

To illustrate how the methodology selects different HwPC sets for individual TP

predictions, Table 5.34 presents the reduced HwPC sets for predicting the WPTX TP for the `Convolution` kernel. It defines the number of elements processed in the X-dimension by each thread (work-item coarsening parameter). The reduced HwPC sets for this TP reveal both commonalities and architecture-specific patterns. Unlike kernel identification, which predominantly relies on HwPCs of type $HwPC_{Ops}$, WPTX prediction incorporates utilization metrics (`INST_ISSUE_U`, `dram_write_throughput`, `gld_requested_throughput`), reflecting the importance of resource saturation in determining optimal work distribution. Instruction-related HwPCs are the most consistent across generations, with `INST_INT` appearing in five architectures (all except Pascal) and `INST_MISC` in four, suggesting that the instruction mix is a fundamental predictor of optimal workload coarsening. However, the evolution of memory-related HwPCs shows a clear architectural progression. Older architectures (Kepler and Maxwell) rely on local memory transactions (`local_load_transactions`, `local_store_transactions`) and basic DRAM metrics, while newer architectures (Turing, Ampere, Ada Lovelace) converge on `LOC_0`, a unified local memory HwPC introduced in the Volta+ profiling API. This shift reflects the improved profiling granularity in newer GPUs. Pascal again exhibits a unique profile, being the only architecture to use `l2_tex_write_transactions` and `dram_write_throughput`, emphasizing L2 cache and DRAM write behavior rather than local memory metrics. The reduced sets remain compact, ranging from 3 to 6 HwPCs per architecture, demonstrating that the methodology efficiently identifies the minimal set of predictive features even for fine-grained TP optimization.

5.5 Discussion of the Evaluation Results

The evaluation results demonstrated both strengths and limitations of the ensemble methodology for optimizing OpenMP code-region TPs. The methodology predicted near-optimal number of threads across diverse code-regions, with K-fold accuracy scores frequently exceeding 0.90 and ROC AUC scores consistently exceeding 0.99. However, performance varied across code-regions, with operations such as `Copy_E` and `Matxvec` showing lower accuracy scores (0.9260 and 0.9041,

5. Evaluation of the Proposed Methodology

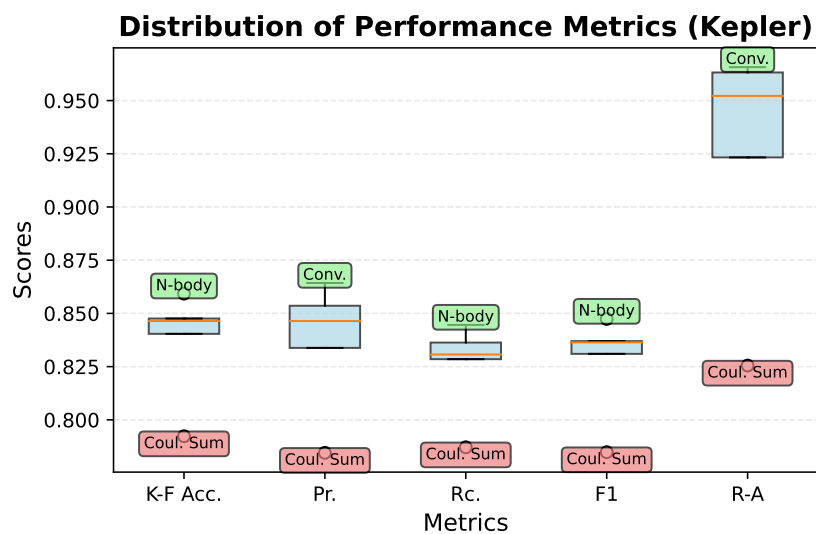


Figure 5.17: Averaged evaluation results of the ensemble for the prediction of the TPs for the Kepler GPU.

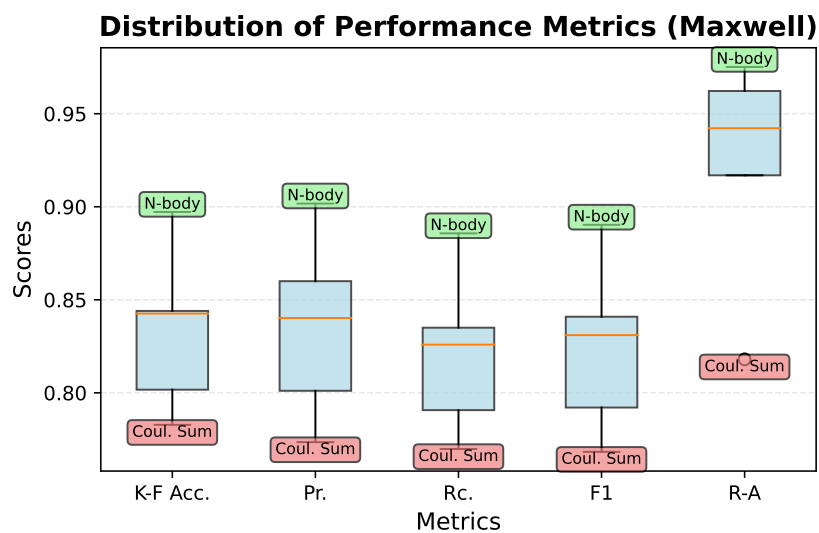


Figure 5.18: Averaged evaluation results of the ensemble for the prediction of the TPs for the Maxwell GPU.

5. Evaluation of the Proposed Methodology

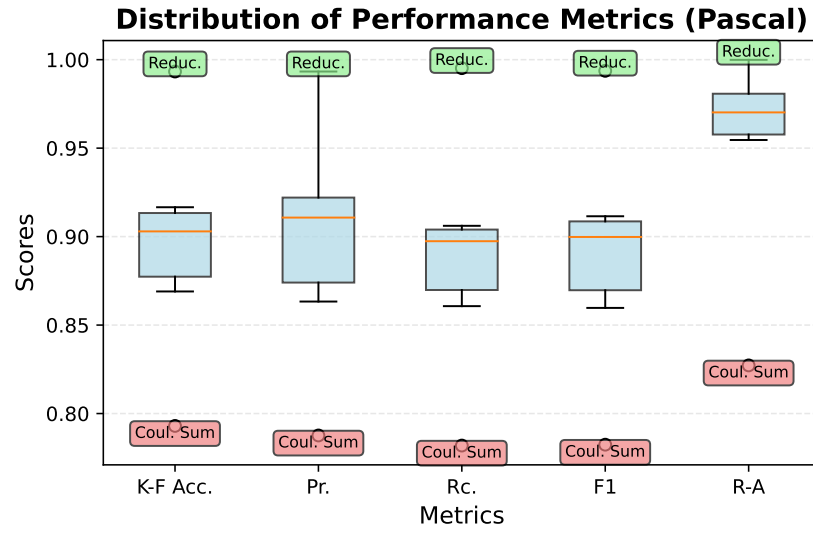


Figure 5.19: Averaged evaluation results of the ensemble for the prediction of the TPs for the Pascal GPU.

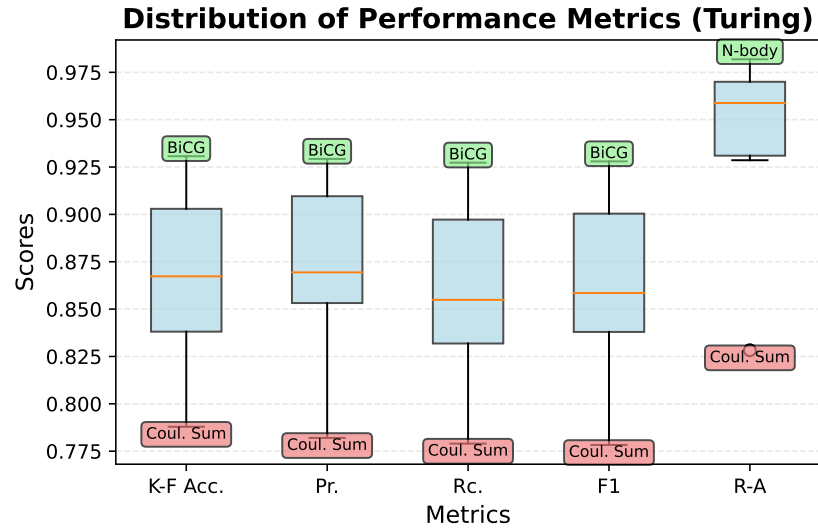


Figure 5.20: Averaged evaluation results of the ensemble for the prediction of the TPs for the Turing GPU.

5. Evaluation of the Proposed Methodology

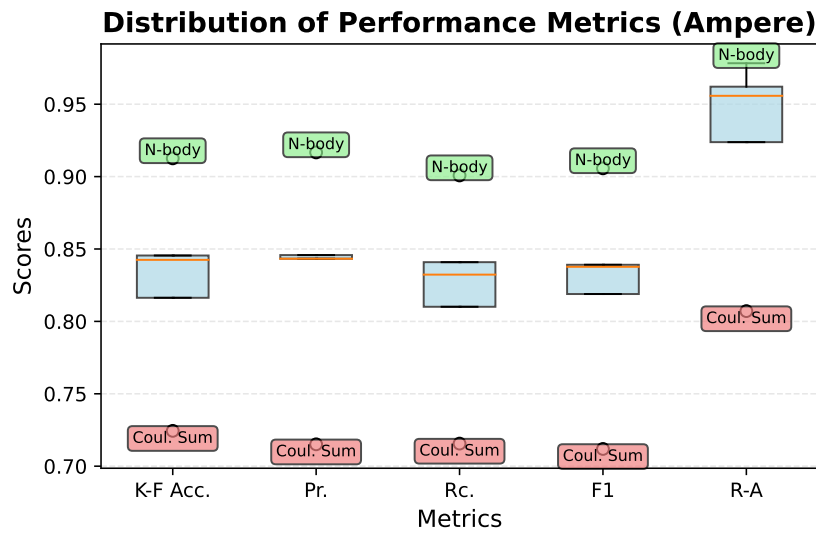


Figure 5.21: Averaged evaluation results of the ensemble for the prediction of the TPs for the Ampere GPU.

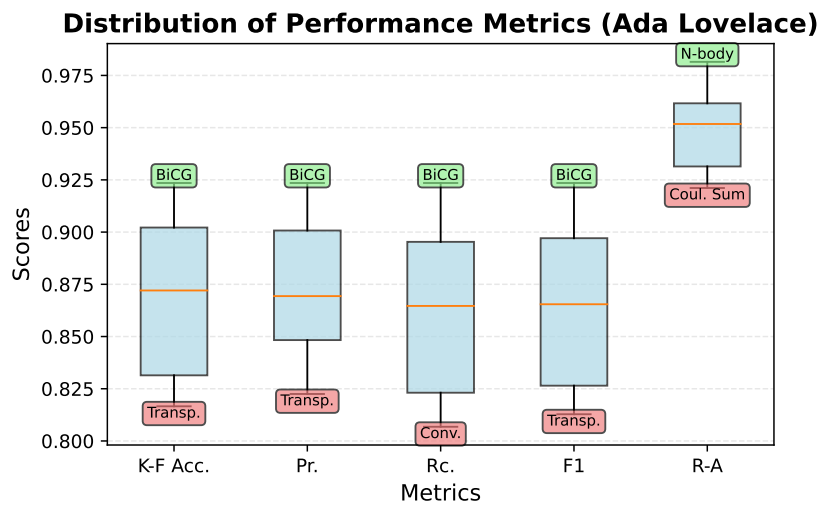


Figure 5.22: Averaged evaluation results of the ensemble for the prediction of the TPs for the Ada Lovelace GPU.

5. Evaluation of the Proposed Methodology

Table 5.33: Kernel TP evaluation average results on the GPUs.

Architecture	Kernel / Metric	K-Fold Accuracy	Precision	Recall	F1-Score	ROC AUC	Reduced
Kepler	Conv.	0.8464	0.8643	0.8285	0.8370	0.9657	2-7
	Coul. Sum	0.7923	0.7845	0.7871	0.7848	0.8255	2-8
	N-body	0.8593	0.8536	0.8446	0.8474	0.9632	2-5
	Transp.	0.8404	0.8338	0.8363	0.8310	0.9522	2-7
	GEMM	0.8476	0.8464	0.8307	0.8364	0.9233	3-9
Maxwell	Conv.	0.8440	0.8600	0.8350	0.8409	0.9622	3-8
	Coul. Sum	0.7828	0.7735	0.7698	0.7683	0.8180	2-9
	N-body	0.8972	0.9017	0.8857	0.8903	0.9751	2-8
	Transp.	0.8017	0.8011	0.7907	0.7921	0.9422	4-8
	GEMM	0.8426	0.8402	0.8259	0.8310	0.9169	3-7
Pascal	Conv.	0.9024	0.9243	0.9061	0.9115	0.9832	2-8
	Coul. Sum	0.7930	0.7876	0.7819	0.7824	0.8272	2-8
	N-body	0.9166	0.9153	0.8976	0.8999	0.9670	2-8
	Transp.	0.8690	0.8633	0.8607	0.8597	0.9734	6-11
	GEMM	0.9035	0.9062	0.8972	0.8997	0.9546	3-11
	Reduc.	0.9933	0.9933	0.9952	0.9937	0.9999	2
Turing	Conv.	0.8351	0.8694	0.8244	0.8383	0.9660	4-8
	Coul. Sum	0.7879	0.7820	0.7790	0.7783	0.8283	4-9
	N-body	0.9200	0.9224	0.9085	0.9129	0.9819	3-6
	Transp.	0.8411	0.8412	0.8393	0.8376	0.9588	5-8
	GEMM	0.8673	0.8652	0.8549	0.8585	0.9334	3-8
	BiCG	0.9307	0.9293	0.9273	0.9280	0.9286	3-9
	Hotspot	0.8859	0.8968	0.8860	0.8879	0.9740	4-8
Ampere	Conv.	0.8163	0.8433	0.8101	0.8189	0.9621	3-7
	Coul. Sum	0.7244	0.7151	0.7156	0.7119	0.8071	2-7
	N-body	0.9126	0.9167	0.9008	0.9057	0.9783	3-6
	Transp.	0.8425	0.8433	0.8409	0.8391	0.9558	5-7
	GEMM	0.8455	0.8458	0.8323	0.8377	0.9238	4-7
Ada Lovelace	Conv.	0.8197	0.8421	0.8068	0.8142	0.9642	2-7
	Coul. Sum	0.8775	0.8718	0.8675	0.8676	0.9211	5-7
	N-body	0.9104	0.9104	0.9046	0.9069	0.9815	2-9
	Transp.	0.8166	0.8225	0.8102	0.8128	0.9495	4-7
	GEMM	0.8666	0.8669	0.8618	0.8633	0.9540	3-6
	BiCG	0.9235	0.9235	0.9235	0.9235	0.9254	2-9

respectively), indicating potential areas for improvement.

Thread affinity prediction demonstrated strong performance, with most code-regions achieving accuracy scores greater than 0.95 while requiring only 3-7 Hw-PCs, suggesting effective optimization with minimal overhead. Compared to OpenTuner, the ensemble methodology demonstrated a crucial advantage in optimization time. While OpenTuner required higher optimization time—notably 32.91

5. Evaluation of the Proposed Methodology

Table 5.34: Reduced HwPCs for predicting WPTX TP for the `Convolution` kernel across GPU architectures.

Type	HwPC (Abbr.)	Cat.	GTX 680 (Kepler)	GTX 750 (Maxwell)	GTX 1070 (Pascal)	RTX 2080 (Turing)	RTX 3090 (Ampere)	RTX 4080 (Ada Lovelace)
Instr.	INST_INT	Ops.	×	×		×	×	×
	INST_MISC	Ops.	×	×	×			×
	INST_ISSUE_U	Ops.	×		×	×	×	×
DRAM	DRAM_RT	Ops.	×		×			
	dram_write_throughput*	Stress			×			
L2 Cache	l2_tex_write_transactions*	Ops.			×			
Local Memory	LOC_0	Ops.				×	×	×
	local_load_transactions*	Ops.		×				
	local_store_transactions*	Ops.		×				
Global Memory	TEX_RWT	Ops.				×		
	gld_requested_throughput*	Stress			×			

* HwPC name not in equivalence Table [23]; original name retained.

seconds for `MatrixMult` and 1 minute 33 seconds for `MatrixMultN0pt`—the ensemble methodology consistently completed optimization in less time across all evaluated code-regions.

The methodology demonstrated versatility in handling other OpenMP optimization aspects, with strong performance in predicting scheduling policies and chunk sizes for the `Collatz` and `Friends` code-regions (accuracies above 0.90). However, limitations include varying performance across code-regions and certain complex computational patterns, indicating the current HwPC set may not fully capture all relevant program behavior aspects. Results highlight a trade-off between optimization quality and speed—while not always achieving the absolute peak performance of search methods like OpenTuner, the ensemble methodology delivers substantial speedups with minimal optimization overhead, making it suitable for dynamic optimization scenarios requiring rapid adaptation to changing code-regions.

Application to NAS Parallel Benchmarks further validated the methodology’s effectiveness across scientific applications code-regions, maintaining strong performance

5. Evaluation of the Proposed Methodology

in code-region identification with most code-regions showing accuracy above 0.90.

While the ensemble methodology generally achieved near-optimal speedups for number of threads and maintained reasonable accuracy in thread affinity prediction, its most significant advantage was computational efficiency. The ensemble consistently completed optimization tasks in seconds compared to OpenTuner’s requirements of minutes to hours, representing execution time improvements of up to 30× while delivering comparable or superior optimization quality.

The GPU validation experiments on GTX 680 (Kepler), GTX 750 (Maxwell), GTX 1070 (Pascal), RTX 2070 (Turing), RTX 3070 (Ampere), and RTX 4070 (Ada Lovelace) architectures demonstrated the methodology’s cross-platform effectiveness across six GPU generations. TP optimization across multiple kernels showed consistently strong performance (>0.80 accuracy, >0.77 precision, and >0.77 recall), with Pascal achieving the highest performance, particularly in specialized kernels like `Reduction` (0.9933 accuracy). Maxwell demonstrated robust optimization capabilities with notable strength in `N-body` (0.8972 accuracy), while Turing and Ada Lovelace showed competitive results across diverse kernels. Ampere showed solid performance across most kernels with some variations in specific kernels. The methodology required only 2-11 HwPCs for most kernels, validating efficient feature utilization and broad applicability across modern GPU architectures spanning six generations in heterogeneous computing environments.

Notably, the majority of HwPCs automatically selected by the ensemble correspond directly to entries in Filipović et al.’s equivalence table [23] — 10 out of 15 for kernel identification and 6 out of 11 for WPTX TP prediction — suggesting that the methodology independently converges on the same discriminative HwPCs previously identified through expert analysis.

Finally, we analyze the computational cost of our methodology. The initial reduction phase, which involves automated HwPC selection and model training, incurs a high but one-time cost per target architecture. The application execution phase, which occurs frequently, introduces minimal overhead. This overhead consists of HwPC data collection and model inference, and has been measured on the same

5. Evaluation of the Proposed Methodology

platform used in the experimentation (Xeon E5-4620), obtaining:

1. Code-region Identification:
 - Collection: ~ 3 it./region, $\bar{t} = 0.1063ms/iter$ ($\sigma = 0.0279ms$)
 - Inference: $\bar{t} = 0.5667ms$ ($\sigma = 0.0270ms$).
2. Tuning Parameter (TP) Optimization:
 - Collection: $\sim 2-3$ it./region, $\bar{t} = 0.0942ms/iter$ ($\sigma = 0.0293ms$).
 - Inference: $\bar{t} = 0.8404ms$ ($\sigma = 0.0122ms$).

The results demonstrate that the runtime overhead of our methodology during application execution is low, making it suitable even for dynamic tuning.

Chapter 6

Conclusions and Future Work

This chapter is divided into two parts: first, the conclusions and main results of the work presented in this thesis are summarized, and secondly, the future work where several open lines of research are discussed.

6.1 Conclusions

This work has researched the possibility of using ML techniques combined with HwPCs to automatically optimize the performance of parallel applications across heterogeneous computing environments. While ML approaches have been applied to performance optimization in other works, to the best of our knowledge, this is the first time a comprehensive methodology has been systematically defined to address both the automated selection of informative HwPCs and the development of efficient optimization strategies for parallel applications, with particular emphasis on portability across diverse architectures and practical applicability in real-world scenarios.

When using ML to generate performance models, the fundamental principle of "garbage in, garbage out" remains crucial. According to this principle, the model will only be as effective as the data and methodologies used in its development, so proper approaches for feature selection, data collection, and optimization strategy

6. Conclusions and Future Work

design are necessary to build useful and reliable models. In the case of automated performance optimization of parallel applications, the following key questions would need to be answered: 1) given the large number of measurable HwPCs available in modern systems, how could application behavior be accurately characterized using a minimal yet informative subset of metrics while maintaining portability across architectures?; 2) how could effective optimization strategies be developed that consider the complex trade-offs between individual region optimization and overall application performance?; and 3) how could this methodology be validated and demonstrated to work effectively across diverse computational patterns and hardware platforms? The methodology developed in this thesis systematically addressed each of these questions through comprehensive research and experimental validation.

The major contribution of this thesis is an automated ML methodology for performance optimization in heterogeneous HPC environments that addresses critical challenges in utilizing HwPCs for application tuning. This methodology integrates HwPC reduction, parallel code-region identification, and TP optimization while demonstrating efficiency and portability across architectures. Specifically, the methodology addresses three fundamental challenges that motivate this work:

1. **Monitoring Limitations:** Hardware limitations prevent simultaneous access to all HwPCs. Measuring fewer, carefully selected events is crucial for accuracy, allowing multiple executions over selected HwPC groups to achieve higher measurement precision than multiplexing-based approaches.
2. **Scalability and Data Volume Constraints:** The volume of HwPC data, especially from GPUs, with hundreds of associated HwPCs, implies serious scalability challenges. Manual analysis becomes impractical, necessitating automated approaches.
3. **Runtime Decision Requirements:** HPC environments require rapid analysis. This demands efficient performance models that use a minimal set of HwPCs that accurately represent application behavior to enable runtime optimization decisions.

Our initial hypothesis was that parallel application performance could be effectively optimized through automated analysis of HwPCs combined with ML techniques, while considering the practical constraints of overhead minimization and architectural portability. To validate this hypothesis, the proposed methodology underwent extensive development and evaluation across multiple hardware platforms and diverse computational workloads, organized around four specific objectives.

Objective 1: Representative Dataset Generation. Chapter 4 presents a methodology for constructing training datasets that capture the diverse behaviors of parallel applications across different execution conditions. The dataset encompasses 18 code-regions from three benchmark suites — STREAM, PolyBench, and irregular computation kernels — covering memory-bound, compute-bound, strided-access, and irregular control-flow patterns. Each configuration was repeated 100 times for statistical significance, yielding 20,044,800 total executions. This objective was addressed in the following publication:

Design of Hardware Performance Counters Collection for Dataset Generation. In: Actas de las Jornadas SARTECO 2022: Avances en arquitectura y tecnología de computadores, Universidad de Alicante, Alicante, Spain, pp. 85–92, 2022. Harutyunyan Gevorgyan, S., César, E., Sikora, A. [27]

Objective 2: Effective HwPC Utilization and Analysis. Chapter 4 also develops an automatic methodology for reducing the number of HwPCs required for both code-region identification and TP optimization. Using a four-model ensemble combined with weighted importance scoring and knee-point detection, the methodology identifies minimal HwPC subsets that preserve predictive accuracy while minimizing overhead. On the Xeon E5-4620, the methodology reduced 50 HwPCs to only 6 for code-region identification — an 88% reduction — achieving 97.65% classification accuracy and a ROC AUC of 0.9996. On the Xeon E5645, it reduced 58 HwPCs to 5, achieving 99.20% accuracy with less than one quarter of the HwPCs required by the prior approach [4]. This objective was addressed in the following publication:

Efficient Code-Region Characterization Through Automatic Performance Counters Reduction Using Machine Learning Techniques. In: Euro-Par 2024: Parallel Processing, Springer Nature Switzerland, Cham, pp. 18–32, 2024. Harutyunyan, S., César, E., Sikora, A., Filipovič, J., Dutta, A., Jannesari, A., Alcaraz, J. [29].

Objective 3: Portable Optimization Methodology. Chapter 5 validates the methodology across seven hardware platforms: one CPU (Intel Xeon E5-4620) and six GPU architectures spanning three NVIDIA generations (GTX 680, GTX 750, GTX 1070, RTX 2080, RTX 3090, and RTX 4080). On the CPU, TP optimization achieved cross-validated scores from 0.9041 to 0.9974 using only 5–6 HwPCs per task, with competitive results against OpenTuner at dramatically reduced optimization time. GPU evaluation confirmed portability across architectures with substantially different HwPC spaces and computational characteristics. This objective was addressed in the following publications:

Automatic Tuning based on Hardware Performance Counters and Machine Learning. Future Generation Computer Systems, vol. 179, p. 108358, 2026. Harutyunyan Gevorgyan, S., César, E., Sikora, A., Filipovič, J., Alcaraz, J. [30].

Complementing this, in a collaborative work, HwPCs are used to analyze and refine tuning spaces — identifying inefficient configurations and uncovering missing optimization opportunities that improve performance in unseen environments. By analyzing how each TP affects HwPCs across the entire tuning space, hidden inefficiencies and interactions between TPs that traditional profiling tools cannot easily reveal are exposed. This HwPC-guided tuning space refinement was validated through a case study on a GPU N-body simulation, improving average configuration performance by $3.3\times$ and the best-performing configuration by 2–18%. This was published in:

Towards Analysis and Refinement of Autotuning Spaces. Parallel Computing, vol. 127, p. 103185, 2026. Filipovič, J., Harutyunyan Gevorgyan, S., César, E., Sikora, A. [22].

Objective 4: End-to-End Automation. Chapter 4 delivers a complete two-phase automated pipeline requiring no manual intervention at runtime. The of-

fine **HwPC Reduction Phase** trains the ensemble models and identifies minimal HwPC subsets for both code-region classification and TP optimization. The online **Application Execution Phase** collects only the reduced HwPCs at runtime, applies majority voting to identify the executing code-region, and invokes region-specific TP optimization models. As demonstrated in [30], this end-to-end automation addresses the gap in prior work by enabling simultaneous code-region identification and TP optimization across diverse architectures. This objective was further supported by the following publication, previously introduced in Objective 3:

Automatic Tuning based on Hardware Performance Counters and Machine Learning. Future Generation Computer Systems, vol. 179, p. 108358, 2026. Harutyunyan Gevorgyan, S., César, E., Sikora, A., Filipovič, J., Alcaraz, J. [30]

Taken together, these results confirm the initial hypothesis of this thesis. Parallel application performance can be effectively optimized through automated analysis of a minimal set of HwPCs combined with an ensemble of ML techniques, while respecting the practical constraints of measurement overhead and architectural portability. The proposed methodology eliminates the manual effort required for performance analysis, reduces monitoring overhead by up to 88% compared to using all available HwPCs, and maintains high predictive accuracy across diverse computational patterns and hardware platforms. The systematic integration of automatic HwPC reduction, representative dataset construction, portable model design, and end-to-end automation offers a comprehensive solution that advances the state-of-the-art in automated performance tuning for modern HPC environments.

Nevertheless, it would be simple, simplistic, and even worse, anachronistic to regard these contributions as a definitive answer to the challenges of performance optimization in HPC environments. The field is evolving rapidly, with new architectural paradigms, emerging workload characteristics, and novel optimization challenges constantly arising. The presented methodology offers a solid foundation for current systems, but must continue to evolve alongside the computing landscape to remain relevant and effective.

6.2 Future Work

This work opens several promising directions for future research. Beyond direct extensions of the methodology, several broader areas also warrant investigation:

1. **Dynamic Adaptation:** The ensemble methodology can be integrated into a dynamic tuning environment that monitors parallel application execution and continuously adjusts TPs during runtime. While the application is running, the online phase collects the reduced set of HwPCs, identifies the currently executing code-region, and invokes the corresponding TP optimization model to infer and apply optimal configurations — all without interrupting execution. Incremental model updates — where models are progressively refined using newly collected runtime data without full retraining — enable the ensemble to adapt to execution phase changes while keeping the impact on application execution negligible. This ensures that both the HwPC collection and the ensemble inference introduce minimal overhead, preserving the low-cost and automated nature of the methodology.
2. **Verification and Validation:** The ensemble methodology can be verified on a wider range of parallel applications to assess its generalizability beyond the classes of code-regions represented by the benchmarks evaluated in this work. Applying the ensemble across diverse application domains and execution patterns allows systematic evaluation of its robustness in dynamically identifying relevant HwPCs and producing reliable optimization decisions under varying runtime conditions.
3. **Multi-objective Optimization:** The ensemble methodology can be extended to support multi-objective optimization by incorporating multiple performance metrics into the learning and decision processes. In this context, different models within the ensemble can capture trade-offs between execution time, energy consumption, and resource utilization, enabling balanced optimization decisions that reflect the constraints and objectives of modern HPC systems.
4. **Joint Tuning Parameter Optimization:** The current methodology op-

timizes each TP independently, using a dedicated model per TP per code-region. A natural extension would be to develop models that jointly optimize multiple TPs simultaneously, capturing the interactions and dependencies between them. This could improve optimization quality in cases where the optimal value of one TP depends on the value of another, such as the interplay between number of threads and thread affinity policy, or between scheduling policy and chunk size.

5. **Extended Architecture Support:** Evaluation of the methodology on non-NVIDIA GPU platforms (e.g., AMD and Intel GPUs) to assess the generalizability of the approach across diverse hardware vendors and generations. Future work could also extend support to emerging architectures such as FPGAs and specialized AI accelerators (ASICs), which would require investigating architecture-specific HwPCs and developing appropriate ML methodologies for these platforms.
6. **Alternative GPU Optimization:** Apply the proposed methodology for different purposes for GPU platforms, such as developing dedicated visualization tools — as preliminarily explored in [22] — that help developers understand how different TPs affect HwPC behavior and overall application performance. This could also include investigating alternative ML methodologies and optimization strategies specifically tailored to GPU architectures and their unique threading models.
7. **Scalability Studies:** Scalability studies are required to evaluate the ensemble methodology in large-scale HPC systems. This includes analyzing the overhead of HwPC data collection, model inference, and decision making as system size increases. Applying the ensemble in hierarchical or distributed tuning frameworks enables performance optimization while maintaining efficiency and practicality at scale.

These research directions extend the ensemble methodology toward dynamic, multi-objective, and large-scale optimization scenarios, expanding its capabilities while maintaining the core design principles of automation, portability, and low overhead

6. Conclusions and Future Work

as HPC systems evolve in complexity and heterogeneity.

References

- [1] L. Adhianto, S. Banerjee, M. Fagan, M. Krentel, G. Marin, J. Mellor-Crummey, and N. R. Tallent. HPCTOOLKIT: tools for performance analysis of optimized parallel programs. *Concurr. Comput.: Pract. Exper.*, 22(6):685–701, April 2010. 15
- [2] Advanced Micro Devices, Inc. AMD EPYC 9654 Processor Specifications. <https://www.amd.com/en/products/processors/server/epyc/4th-generation-9004-and-8004-series/amd-epyc-9654.html>. Accessed: 2025-10-20. 2
- [3] Jordi Alcaraz. *Towards Automatic Generation of Performance Models for Dynamic Tuning using Machine Learning*. PhD dissertation, Universitat Autònoma de Barcelona, 2021. 42, 43
- [4] Jordi Alcaraz, Anna Sikora, and Eduardo César. Hardware Counters’ Space Reduction for Code-Region Characterization. In Ramin Yahyapour, editor, *Euro-Par 2019: Parallel Processing*, pages 74–86, Cham, 2019. Springer International Publishing. 39, 42, 69, 87, 88, 133
- [5] Jordi Alcaraz, Ali Tehrani Jamsaz, Akash Dutta, Anna Sikora, Ali Jannesari, Joan Sorribes, and Eduardo César. Predicting Number of Threads using Balanced Datasets for OpenMP regions. *PDP Special Issue 2021 in Computing*, 105(5):999–1017, 2023. 40, 43, 91
- [6] Jason Ansel, Shoaib Kamil, Kalyan Veeramachaneni, Jonathan Ragan-Kelley, Jeffrey Bosboom, Una-May O’Reilly, and Saman Amarasinghe. OpenTuner:

REFERENCES

- an extensible framework for program autotuning. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation Techniques (PACT)*, PACT '14, pages 303—316, New York, NY, USA, Aug 2014. Association for Computing Machinery. 18, 90
- [7] Sercan Ömer Arik and Tomas Pfister. TabNet: Attentive Interpretable Tabular Learning. *CoRR*, abs/1908.07442, 2019. 51, 66
- [8] D.H. Bailey, E. Barszcz, J.T. Barton, D.S. Browning, R.L. Carter, L. Dagum, R.A. Fatoohi, P.O. Frederickson, T.A. Lasinski, R.S. Schreiber, H.D. Simon, V. Venkatakrishnan, and S.K. Weeratunga. The NAS Parallel Benchmarks. *The International Journal of Supercomputing Applications*, 5(3):63–73, 1991. 87, 107
- [9] David A. Beckingsale, Jason Burmark, Rich Hornung, Holger Jones, William Killian, Adam J. Kunen, Olga Pearce, Peter Robinson, Brian S. Ryujin, and Thomas RW Scogland. RAJA: Portable Performance for Large-Scale Scientific Applications. In *2019 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*, pages 71–81, 2019. 24
- [10] C.M. Bishop. *Pattern Recognition and Machine Learning*. Information Science and Statistics. Springer New York, 2016. 82
- [11] Leo Breiman. Random Forests. *Machine Learning*, 45:5–32, jan 2001. 47, 66
- [12] François Broquedis, Jérôme Clet-Ortega, Stéphanie Moreaud, Nathalie Furmento, Brice Goglin, Guillaume Mercier, Samuel Thibault, and Raymond Namyst. hwloc: a Generic Framework for Managing Hardware Affinities in HPC Applications. In IEEE, editor, *PDP 2010 - The 18th Euromicro International Conference on Parallel, Distributed and Network-Based Computing*, Pisa, Italy, February 2010. 68
- [13] S. Browne, J. Dongarra, N. Garner, K. London, and P. Mucci. A Scalable Cross-Platform Infrastructure for Application Performance Tuning Using Hardware Counters. In *SC '00: Proceedings of the 2000 ACM/IEEE Conference on Supercomputing*, pages 42–42, 2000. 42

- [14] Alexandru Calotoiu, Torsten Hoefler, Marius Poke, and Felix Wolf. Using Automated Performance Modeling to Find Scalability Bugs in Complex Codes. In *Proc. of the ACM/IEEE Conference on SC13, Denver, CO, USA*, pages 1–12. ACM, November 2013. 17
- [15] E. Cesar, A. Moreno, J. Sorribes, and E. Luque. Modeling Master/Worker applications for automatic performance tuning. *Parallel Computing*, 32(7):568–589, 2006. Algorithmic Skeletons. 94
- [16] Tianqi Chen and Carlos Guestrin. XGBoost: A Scalable Tree Boosting System. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD '16*, pages 785–794, New York, NY, USA, 2016. ACM. 47, 51, 66
- [17] David R. Cox. The regression analysis of binary sequences. *Journal of the Royal Statistical Society: Series B (Methodological)*, 20(2):215–232, 1958. 45, 65
- [18] L. Dagum and R. Menon. OpenMP: an industry standard API for shared-memory programming. *IEEE Computational Science and Engineering*, 5(1):46–55, 1998. 13
- [19] Alejandro Duran, Xavier Teruel, Roger Ferrer, Xavier Martorell, and Eduard Ayguade. Barcelona OpenMP Tasks Suite: A Set of Benchmarks Targeting the Exploitation of Task Parallelism in OpenMP. In *Proceedings of the 2009 International Conference on Parallel Processing, ICPP '09*, pages 124–131, USA, 2009. IEEE Computer Society. 30, 35
- [20] Akash Dutta, Jordi Alcaraz, Ali Tehrani Jamsaz, Anna Sikora, Eduardo Cesar, and Ali Jannesari. Pattern-based Autotuning of OpenMP Loops using Graph Neural Networks. In *2022 IEEE/ACM International Workshop on Artificial Intelligence and Machine Learning for Scientific Applications (AI4S)*, pages 26–31, 2022. 31
- [21] H. Carter Edwards, Christian R. Trott, and Daniel Sunderland. Kokkos: Enabling manycore performance portability through polymorphic memory ac-

REFERENCES

- cess patterns. *Journal of Parallel and Distributed Computing*, 74(12):3202 – 3216, 2014. Domain-Specific Languages and High-Level Frameworks for High Performance Computing. 24
- [22] Jiří Filipovič, Suren Harutyunyan Gevorgyan, Eduardo César, and Anna Sikora. Towards analysis and refinement of autotuning spaces. *Parallel Computing*, 127:103185, 2026. 134, 137
- [23] Jiří Filipovič, Jana Hozzová, Amin Nezarat, Jaroslav Ol’ha, and Filip Petrovič. Using hardware performance counters to speed up autotuning convergence on GPUs. *Journal of Parallel and Distributed Computing*, 160:16–35, 2022. 28, 118, 119, 127, 128
- [24] Peter I. Frazier. Bayesian Optimization. *Recent Advances in Optimization and Modeling of Contemporary Problems*, 2018. 23
- [25] Markus Geimer, Felix Wolf, Brian J. N. Wylie, Erika Ábrahám, Daniel Becker, and Bernd Mohr. The Scalasca performance toolset architecture. *Concurr. Comput.: Pract. Exper.*, 22(6):702—719, April 2010. 15, 16, 17
- [26] Michael Gerndt, Karl Förlinger, and Edmond Kereku. Periscope: Advanced Techniques for Performance Analysis. In Gerhard R. Joubert, Wolfgang E. Nagel, Frans J. Peters, Oscar G. Plata, P. Tirado, and Emilio L. Zapata, editors, *Parallel Computing: Current & Future Issues of High-End Computing, Proceedings of the International Conference ParCo 2005, 13-16 September 2005, Department of Computer Architecture, University of Malaga, Spain*, volume 33 of *John von Neumann Institute for Computing Series*, pages 15–26. Central Institute for Applied Mathematics, Jülich, Germany, 2005. 14
- [27] Suren Harutyunyan Gevorgyan, Eduardo César, and Anna Sikora. Design of Hardware Performance Counters Collection for Dataset Generation. In Arturo Gonzalez-Escribano, Diego R. Llanos, Jose Manuel Palomares Muñoz, Jorge Azorín-López, Jose Luis Sánchez Romero, and Antonio Martínez-Álvarez, editors, *Actas de las Jornadas SARTECO 2022: Avances en arquitectura y tecnología de computadores*, pages 85–92, Alicante, Spain, September 2022.

- Universidad de Alicante. Jornadas SARTECO 2022, 21–23 September 2022. 133
- [28] Gavin Hackeling. *Mastering Machine Learning With scikit-learn*. Packt Publishing, 2014. 44
- [29] Suren Harutyunyan, Eduardo César, Anna Sikora, Jiří Filipovič, Akash Dutta, Ali Jannesari, and Jordi Alcaraz. Efficient Code-Region Characterization Through Automatic Performance Counters Reduction Using Machine Learning Techniques. In Jesus Carretero, Sameer Shende, Javier Garcia-Blas, Ivona Brandic, Katzalin Olcoz, and Martin Schreiber, editors, *Euro-Par 2024: Parallel Processing*, pages 18–32, Cham, 2024. Springer Nature Switzerland. 7, 90, 134
- [30] Suren Harutyunyan Gevorgyan, Eduardo César, Anna Sikora, Jiří Filipovič, and Jordi Alcaraz. Automatic Tuning based on Hardware Performance Counters and Machine Learning. *Future Generation Computer Systems*, 179:108358, 2026. 8, 79, 134, 135
- [31] T. Hastie, R. Tibshirani, and J.H. Friedman. *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*. Springer series in statistics. Springer, 2009. 82
- [32] Tin Kam Ho. Random decision forests. In *Proceedings of 3rd international conference on document analysis and recognition*, volume 1, pages 278–282. IEEE, 1995. 47
- [33] Arthur E. Hoerl and Robert W. Kennard. Ridge Regression: Biased Estimation for Nonorthogonal Problems. *Technometrics*, 12(1):55–67, 1970. 45, 65
- [34] Intel Corporation. Intel Xeon Platinum 8592+ Processor (64-core, 128-thread, 1.90 GHz base) Specifications. <https://www.intel.com/content/www/us/en/products/sku/237261/intel-xeon-platinum-8592-processor-320m-cache-1-90-ghz/specifications.html>. Accessed: 2025-10-20. 2

REFERENCES

- [35] Heike Jagode, Anthony Danalis, Giuseppe Congiu, Daniel Barry, Anthony Castaldo, and Jack Dongarra. Advancements of PAPI for the exascale generation. *The International Journal of High Performance Computing Applications*, 39(2):251–268, 2025. 42
- [36] Ian Karlin, Abhinav Bhatele, Brad Chamberlain, Jonathan Cohen, Zachary Devito, Maya Gokhale, Riyaz Haque, Rich Hornung, Jeff Keasler, Dan Laney, Edward Luke, Scott Lloyd, Jim McGraw, Rob Neely, David Richards, Martin Schulz, Charles H. Still, Felix Wang, and Daniel Wong. LULESH Programming Model and Performance Ports Overview. Technical Report LLNL-TR-608824, Lawrence Livermore National Laboratory (LLNL), 2012. 25
- [37] Per Gunnar Kjeldsberg, Andreas Gocht, Michael Gerndt, Lubomir Riha, Joseph Schuchart, and Umbreen Sabir Mian. READEX: Linking two ends of the computing continuum to improve energy-efficiency in dynamic applications. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017*, pages 109–114, 2017. 21
- [38] Andreas Knüpfer, Holger Brunst, Jens Doleschal, Matthias Jurenz, Matthias Lieber, Holger Mickler, Matthias S. Müller, and Wolfgang E. Nagel. The Vampir Performance Analysis Tool-Set. In Michael Resch, Rainer Keller, Valentin Himmler, Bettina Krammer, and Alexander Schulz, editors, *Tools for High Performance Computing*, pages 139–155, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg. 15
- [39] Andreas Knüpfer, Christian Rössel, Dieter an Mey, Scott Biersdorff, Kai Diethelm, Dominic Eschweiler, Markus Geimer, Michael Gerndt, Daniel Lorenz, Allen Malony, Wolfgang E. Nagel, Yury Oleynik, Peter Philippen, Pavel Saviankou, Dirk Schmidl, Sameer Shende, Ronny Tschüter, Michael Wagner, Bert Wesarg, and Felix Wolf. Score-P: A Joint Performance Measurement Run-Time Infrastructure for Periscope, Scalasca, TAU, and Vampir. In Holger Brunst, Matthias S. Müller, Wolfgang E. Nagel, and Michael M. Resch, editors, *Tools for High Performance Computing 2011*, pages 79–91, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg. 14

- [40] Jonas H. Müller Korndörfer, Ahmed Eleliemy, Ali Mohammed, and Florina M. Ciorba. LB4OMP: A Dynamic Load Balancing Library for Multithreaded Applications. *IEEE Transactions on Parallel and Distributed Systems*, 33(4):830–841, 2022. 25
- [41] Jonas H. Müller Korndörfer, Ali Mohammed, Ahmed Eleliemy, Quentin Guilleau, Reto Krummenacher, and Florina M. Ciorba. Learning To Select Scheduling Algorithms in OpenMP. Poster presented at SC25: The International Conference for High Performance Computing, Networking, Storage, and Analysis, November 2025. November 16–21, 2025. 25
- [42] Lawrence Livermore National Laboratory and collaborators. CleverLeaf: An Adaptive Mesh Refinement Hydrodynamics Mini-Application. <https://github.com/UK-MAC/CleverLeaf>, 2016. Extends CloverLeaf with Adaptive Mesh Refinement using SAMRAI. 25
- [43] Zhen Li, Rohit Atre, Zia Ul-Huda, Ali Jannesari, and Felix Wolf. DiscoPoP: A Profiling Tool to Identify Parallelization Opportunities. In Christoph Niethammer, José Gracia, Andreas Knüpfer, Michael M. Resch, and Wolfgang E. Nagel, editors, *Tools for High Performance Computing 2014*, pages 37–54, Cham, 2015. Springer International Publishing. 32
- [44] Kevin London, Shirley Moore, Philip Mucci, Keith Seymour, and Richard Luczak. The PAPI Cross-Platform Interface to Hardware Performance Counters. In *Department of Defense Users’ Group Conference Proceedings*, pages 18–21, June 2001. 42, 69
- [45] Allan Mallinson, John Herdman, David Beckingsale, and Simon McIntosh-Smith. CloverLeaf: A Benchmark for Hydrodynamics Codes. In *Proceedings of the Cray User Group Conference*, 2013. 25
- [46] John McCalpin. Memory bandwidth and machine balance in high performance computers. *IEEE Technical Committee on Computer Architecture Newsletter*, pages 19–25, 12 1995. 40, 73, 88, 90

REFERENCES

- [47] Renato Miceli, Gilles Civario, Anna Sikora, Eduardo César, Michael Gerndt, Houssam Haitof, Carmen Navarrete, Siegfried Benkner, Martin Sandrieser, Laurent Morin, and François Bodin. AutoTune: A Plugin-Driven Approach to the Automatic Tuning of Parallel Applications. In Pekka Manninen and Per Öster, editors, *Applied Parallel and Scientific Computing*, pages 328–342, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg. 14
- [48] Robert Mijaković, Michael Firlbach, and Michael Gerndt. An architecture for flexible auto-tuning: The Periscope Tuning Framework 2.0. In *2016 2nd International Conference on Green High Performance Computing (ICGHPC)*, pages 1–9, 02 2016. 14, 21
- [49] Ali Mohammed, Jonas H. Müller Korndörfer, Ahmed Eleliemy, and Florina M. Ciorba. Automated Scheduling Algorithm Selection and Chunk Parameter Calculation in OpenMP. *IEEE Trans. Parallel Distrib. Syst.*, 33(12):4383–4394, December 2022. 25
- [50] Anna Morajko. *Dynamic Tuning of Parallel/Distributed Applications*. PhD thesis, Universitat Autònoma de Barcelona, 2004. 13
- [51] Anna Morajko, Oleg Morajko, Tomàs Margalef, and Emilio Luque. MATE: Dynamic Performance Tuning Environment. In Marco Danelutto, Marco Vanneschi, and Domenico Laforenza, editors, *Euro-Par 2004 Parallel Processing*, pages 98–107, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg. 13
- [52] Philip J. Mucci, Shirley Browne, Christine Deane, and George Ho. PAPI: A Portable Interface to Hardware Performance Counters. In *Proceedings of the Department of Defense HPCMP Users Group Conference*, volume 710, pages 7–10, June 1999. 42, 69
- [53] Jonas H. Müller Korndörfer, Ali Mohammed, Ahmed Eleliemy, Quentin Guilleloteau, Reto Krummenacher, and Florina M. Ciorba. A Comparative Study of OpenMP Scheduling Algorithm Selection Strategies. *IEEE Access*, 13:151216–151234, 2025. 25

- [54] Vojtěch Nikl, Lubomír Říha, Ondřej Vysocký, and Jan Zapletal. Optimal Hardware Parameters Prediction for Best Energy-to-Solution of Sparse Matrix Operations Using Machine Learning Techniques. In *INFOCOMP 2018, The Eighth International Conference on Advanced Communications and Computation*, pages 43–48, Barcelona, 2018. International Academy, Research, and Industry Association. 22
- [55] David Opitz and Richard Maclin. Popular ensemble methods: an empirical study. *J. Artif. Int. Res.*, 11(1):169—198, July 1999. 54, 64
- [56] Filip Petrovič and Jiří Filipovič. Kernel Tuning Toolkit. *SoftwareX*, 22:101385, 2023. 116
- [57] F. Petrovič, D. Střelák, J. Hozzová, J. Olha, R. Trembecký, S. Benkner, and J. Filipovič. A benchmark set of highly-efficient CUDA and OpenCL kernels and its dynamic autotuning with Kernel Tuning Toolkit. *Future Generation Computer Systems*, 108:161–177, 2020. 19, 116
- [58] Vincent Pillet, Jesús Labarta, Toni Cortes, and Sergi Girona. PARAVÉR: A tool to visualize and analyze parallel code. *WoTUG-18*, 44, 03 1995. 15
- [59] R. Polikar. Ensemble based systems in decision making. *IEEE Circuits and Systems Magazine*, 6(3):21–45, 2006. 64
- [60] Mihail Popov, Chadi Akel, Yohan Chatelain, William Jalby, and Pablo de Oliveira Castro. Piecewise Holistic Autotuning of Parallel Programs with CERE. *Concurrency and Computation: Practice and Experience*, 29, 06 2017. 30
- [61] Ahmad Qawasmeh, Abid M. Malik, and Barbara M. Chapman. Adaptive OpenMP Task Scheduling Using Runtime APIs and Machine Learning. In *2015 IEEE 14th International Conference on Machine Learning and Applications (ICMLA)*, pages 889–895, 2015. 29
- [62] Rohan Basu Roy, Tirthak Patel, Vijay Gadepally, and Devesh Tiwari. Bliss: auto-tuning complex applications using a pool of diverse lightweight learn-

REFERENCES

- ing models. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2021, pages 1280—1295, New York, NY, USA, 2021. Association for Computing Machinery. 22
- [63] Fadil Santosa and William W. Symes. Linear Inversion of Band-Limited Reflection Seismograms. *SIAM Journal on Scientific and Statistical Computing*, 7(4):1307–1330, 1986. 45
- [64] Ville Satopaa, Jeannie Albrecht, David Irwin, and Barath Raghavan. Finding a "Kneedle" in a Haystack: Detecting Knee Points in System Behavior. In *2011 31st International Conference on Distributed Computing Systems Workshops*, pages 166–171, 2011. 83
- [65] Joseph Schuchart, Michael Gerndt, Per Gunnar Kjeldsberg, Michael Lysaght, David Horák, Lubomír Zlha, Andreas Gocht, Mohammed Sourouri, Madhura Kumaraswamy, Anamika Chowdhury, Magnus Jahre, Kai Diethelm, Othman Bouizi, Umbreen Sabir Mian, Jakub Kružík, Radim Sojka, Martin Beseda, Venkatesh Kannan, Zakaria Bendifallah, Daniel Hackenberg, and Wolfgang E. Nagel. The READEX formalism for automatic tuning for energy efficiency. *Computing*, 99(8):727—745, August 2017. 21
- [66] Sameer S. Shende and Allen D. Malony. The Tau Parallel Performance System. *Int. J. High Perform. Comput. Appl.*, 20(2):287—311, May 2006. 15
- [67] Robert Tibshirani. Regression Shrinkage and Selection via the Lasso. *Journal of the Royal Statistical Society: Series B (Methodological)*, 58(1):267–288, 1996. 45, 65
- [68] J. Treibig, G. Hager, and G. Wellein. LIKWID: A lightweight performance-oriented tool suite for x86 multicore environments. In *Proceedings of PSTI2010, the First International Workshop on Parallel Software Tools and Tool Infrastructures*, San Diego CA, 2010. 68
- [69] Christian R. Trott, Damien Lebrun-Grandié, Daniel Arndt, Jan Ciesko, Vinh Dang, Nathan Ellingwood, Rahulkumar Gayatri, Evan Harvey, Daisy S. Holl-

- man, Dan Ibanez, Nevin Liber, Jonathan Madsen, Jeff Miles, David Poliakoff, Amy Powell, Sivasankaran Rajamanickam, Mikael Simberg, Dan Sunderland, Bruno Turcksin, and Jeremiah Wilke. Kokkos 3: Programming Model Extensions for the Exascale Era. *IEEE Transactions on Parallel and Distributed Systems*, 33(4):805–817, 2022. 24
- [70] Niki van Stein and Thomas Bäck. LLaMEA: Automatically Generating Meta-heuristics with Large Language Models. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion*, GECCO '25 Companion, pages 81—82, New York, NY, USA, 2025. Association for Computing Machinery. 26
- [71] Ben van Werkhoven. Kernel Tuner: A search-optimizing GPU code auto-tuner. *Future Generation Computer Systems*, 90:347–358, 2019. 20
- [72] Vincent M. Weaver and Sally A. McKee. Can hardware performance counters be trusted? In *2008 IEEE International Symposium on Workload Characterization*, pages 141–150, 2008. 27
- [73] Baptiste Wicht, Roberto A. Vitillo, Dehao Chen, and David Levinthal. Hardware Counted Profile-Guided Optimization, 2014. 12
- [74] Floris-Jan Willemsen, Niki van Stein, and Ben van Werkhoven. Automated Algorithm Design for Autotuning Optimizers. *arXiv preprint arXiv:2510.17899*, October 2025. 26
- [75] Felix Wolf and Bernd Mohr. KOJAK - A Tool Set for Automatic Performance Analysis of Parallel Applications. In *Proc. of the 9th Euro-Par Conference, Klagenfurt, Austria*, volume 2790 of *Lecture Notes in Computer Science*, pages 1301–1304. Springer, August 2003. 16
- [76] Chad Wood, Giorgis Georgakoudis, David Beckingsale, David Poliakoff, Alfredo Gimenez, Kevin Huck, Allen Malony, and Todd Gamblin. Artemis: Automatic Runtime Tuning of Parallel Execution Parameters Using Machine Learning. In *International Conference on High Performance Computing*, pages 453–472. Springer, 2021. 24, 25

REFERENCES

- [77] Akash Yadav and Mushtaq Ahmed. Predictive Modeling for Thread Optimization in OpenMP-Based Parallelization Using Machine Learning. In *2024 IEEE 17th International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSoc)*, pages 339–344, 2024. 23
- [78] T. Yuki and L.-N. Pouchet. Polybench 4.0, 2015. 40, 73, 90
- [79] Tomofumi Yuki. Understanding PolyBench/C 3.2 Kernels. In Sanjay Rajopadhye and Sven Verdoolaege, editors, *Proceedings of the 4th International Workshop on Polyhedral Compilation Techniques*, Vienna, Austria, January 2014. 40, 73, 90
- [80] Hui Zou and Trevor Hastie. Regularization and Variable Selection via the Elastic Net. *Journal of the Royal Statistical Society. Series B (Statistical Methodology)*, 67(2):301–320, 2005. 45, 46, 65