DOI 10.34185/1562-9945-5-161-2025-08
УДК 004.89

M. Berdnyk, I. Starodubskyi

SELF-PROFILING MECHANISMS FOR REAL-TIME CODE COMPILERS

Anotation. This paper explores the concept of self-profiling compilers as a means of adaptive real-time code optimization. The approach relies on collecting dynamic performance metrics during program execution and analyzing the collected data to choose the most effective compilation strategies. We propose a compiler architecture capable of automatically detecting performance-critical code regions (hotspots), adjusting the configuration of optimization passes, and recompiling code with updated metrics. A prototype was implemented based on LLVM with an embedded runtime agent responsible for code instrumentation, metric collection, and interaction with a dynamic Pass Manager. A series of experiments were conducted across various hardware platforms, including desktop CPUs and ARM-based architectures. The results demonstrated significant performance gains without noticeable increases in compilation time or resource usage. These findings confirm the feasibility of integrating self-profiling into next-generation compilers targeting high-performance computing, edge systems, and mobile devices. The paper presents the concept of self-profiling as a tool for real-time code optimization. A prototype based on LLVM with embedded runtime analysis has been implemented. The results demonstrate the advantages of the proposed approach.
Keywords: adaptive compilation; runtime optimization; self-profiling; LLVM; JIT compilation; performance.

Statement of the problem. Traditional compilers implement fixed optimization strategies that do not account for the dynamic nature of program execution in real-world environments. This becomes especially critical in heterogeneous or variable systems, where system load, hardware configuration, or usage scenarios change in real time.

Such limitations lead to inefficient utilization of hardware resources, reduced performance, increased energy consumption, and limited scalability.

In response to these challenges, there is a growing need for compilers capable of self-profiling – dynamically tracking program execution, analyzing collected metrics, and adapting the compilation strategy during runtime.

These mechanisms enable flexible optimization management, reduce manual tuning overhead, and provide high adaptability to new runtime environments.

Such approaches are particularly relevant for resource-constrained systems, mobile devices, edge computing platforms, and server infrastructures with a high degree of parallelism, where efficient use of compute time is critical.

Self-profiling enables localized optimization that takes into account the specifics of a particular environment, task type, and the real-time conditions under which the program executes.

In this context, the development of compilers capable of autonomously detecting “hot” code regions and adjusting their processing without halting or rebuilding the entire application is a key research direction in modern compiler engineering.

Integrating such mechanisms into standard compilation pipelines can significantly reduce the time to reach peak performance and make compilation more flexible, intelligent, and responsive to execution context.

Analysis of the latest research and publications. In recent years, the research community has paid considerable attention to issues of dynamic optimization and the integration of runtime analysis into compilers.

There are several approaches that partially implement self-profiling concepts. For example, GraalVM includes built-in JIT compilation mechanisms that allow modifying program behavior based on collected statistics. However, these systems are primarily focused on interpreted languages (Java, JavaScript) and have limited applicability at the system level or for C++/LLVM-based projects.

Other efforts, such as LLVM Profile-Guided Optimization (PGO), require an initial execution phase with instrumentation followed by recompilation. This does not allow real-time adaptation.

Some recent studies [1], [2] have explored combining machine learning with optimization passes, but still depend on manual data collection or lengthy model training processes.

At the same time, modern compilers such as Clang/LLVM provide the tools to implement custom analysis and transformation passes at the IR level, but lack built-in support for runtime feedback. Approaches based solely on static heuristics often fail to yield optimal results in highly dynamic workloads.

Using runtime profiling as the foundation for adaptive compilation remains a promising research direction, but it requires systematic integration and technological maturity. Some works propose hybrid models that combine offline analysis with partial runtime adaptation, but these do not offer full automation. Projects targeting mobile platforms (e.g., Facebook Hermes) implement lightweight JIT mechanisms, but their capabilities are limited by performance and power constraints.

In the field of high-performance computing (HPC), machine learning is being explored to manage optimization passes based on profiling [3],[4], though such methods require extensive training datasets.

LLVM’s AutoFDO initiative (Automatic Feedback-Directed Optimization) proposes using hardware performance counters to build execution profiles without instrumentation [6]. While this reduces overhead, the granularity of the collected data remains low.

Emerging work at the intersection of compiler infrastructure and neural models—such as Graph Neural Networks (GNN) for representing IR—also shows promising results for predicting transformation utility [4],[10].

ISSN 1562-9945 (Print)
ISSN 2707-7977 (Online)

Another important challenge is accurately identifying the boundary between “hot” and “cold” code regions – most current approaches rely on heuristics or empirical observations, complicating automation.

In this context, developing a compiler capable of collecting, interpreting, and using runtime metrics without developer intervention remains an open challenge [2], [7]. Thus, the task of building a compiler with integrated runtime analysis that can autonomously adapt to changing execution environments remains unresolved and highly relevant. There is a clear need for holistic architectures that tightly integrate profiling, adaptive optimizations, and transformation management into a unified compilation pipeline.

Research Objective. The aim of this study is to design and implement a compiler architecture capable of self-profiling in real time, enabling dynamic adaptation of optimization strategies based on the runtime characteristics of the program code.

This architecture is built around deep integration between runtime execution analysis and the compilation process itself, allowing the compiler to respond intelligently to the specific context in which an application operates.

A core focus is the development of mechanisms that allow the system to collect key performance metrics during execution without any user intervention. These include function call frequency, execution time of specific code blocks, cache utilization, memory allocation frequency, and other behavioral indicators.

Special attention is given to enabling selective recompilation or transformation of only the relevant code segments. Instead of performing full program recompilation, the system applies targeted re-JIT (just-in-time recompilation) in an automated fashion. This ensures both flexibility and high performance.

The architecture is designed to be fully transparent to the developer: all monitoring and adaptation processes are handled in the background, with no need to modify the source code or add special annotations.

One of the key tasks is to develop a flexible mechanism for communication between runtime components and the compilation environment. This includes building an adaptive Pass Manager for LLVM that can dynamically reorganize the sequence of optimization passes based on execution data collected at runtime [1], [2], [6].

Such a mechanism would enable an intelligent compilation strategy where optimization decisions are not predetermined, but derived from profiling results for each specific program run[2], [4].

The ultimate goal is to reduce energy consumption and execution time without compromising stability or compatibility. The resulting system must be scalable and suitable for both high-performance computing scenarios and resource-constrained environments – such as mobile and edge platforms[3], [7].

Thus, by implementing self-profiling mechanisms, we aim to shift from traditional static compilation toward a dynamic, intelligent compilation model that adapts to real-world execution contexts.

Presentation of the Main Research Material. The proposed compiler architecture is based on a modular approach that allows the core components of the self-profiling system to be implemented in isolation.

The main idea is to decouple the compilation and optimization processes into discrete stages, connected via a shared infrastructure for collecting and analyzing performance metrics [1], [2].

The system supports a full adaptive compilation cycle that begins with source code instrumentation and ends with just-in-time (JIT) recompilation of selected fragments during runtime.

The first stage of the compiler pipeline involves inserting logging instructions into the LLVM IR code. A custom LLVM pass was developed to modify the intermediate representation (IR) by injecting profiling function calls at defined points – function entries, conditional branches, basic block terminations, and memory allocation points.

These modifications do not affect program logic but provide precise control over execution flow tracking.

The instrumented code is compiled with support for runtime monitoring, implemented as a background agent written in C++. This agent utilizes multithreading primitives, timing mechanisms, and synchronization constructs from the standard library. It listens to profiling signals emitted from the inserted instrumentation points, aggregates temporal and structural execution data, and transfers the collected metrics to a shared runtime profile buffer.

A hotspot analysis module is used to detect bottlenecks. Its task is to identify IR fragments with the highest load, call frequency, or average execution latency.

Detection is based on sliding averages with exponential smoothing and threshold-based performance deviation tracking over time intervals. This enables the system to identify both constantly overloaded and recently emerged critical regions caused by changing runtime conditions.

Based on this analysis, the compiler dynamically generates a new optimization configuration. A specially extended LLVM Pass Manager is used, capable of adjusting the active pass sequence depending on the IR fragment type, call frequency, and execution behavior patterns.

For example, frequently invoked functions with heavy arithmetic operations are compiled with aggressive passes such as loop unrolling, inlining, global constant propagation, and expression reassociation. Meanwhile, background or I/O functions retain a minimal optimization profile to reduce unnecessary compilation overhead.

The final stage is runtime JIT recompilation of selected functions. LLVM ORC (On Request Compilation) is used to enable dynamic translation of code segments during execution[6], [10].

Optimized versions replace their predecessors in the dispatch table, allowing the program to continue uninterrupted. Function updates are managed via the IRTransformLayer interface, which handles caching of compiled blocks and ensures memory layout consistency within the runtime module.

ISSN 1562-9945 (Print)
ISSN 2707-7977 (Online)

This architecture avoids full recompilation of the entire program, focusing only on segments that have a proven performance impact. The system is fully transparent to the developer: no source code changes or manual configuration are required. All profiling and optimization logic runs automatically based on real-time execution analytics.

Instrumentation Examples. Instrumentation is not implemented as trivial logging calls, but as full-fledged IR instruction injections integrated with the self-profiling system through a shared memory buffer or feedback API. These methods are designed to capture execution time, loop iteration counts, and memory access patterns with minimal performance impact.

Measuring Function Execution Time. For each target function in the IR, a pair of instructions—start_time and end_time—is inserted. These retrieve high-resolution timestamps (e.g., via rdtscp) and compute execution latency with near-cycle-level precision.

```
// Inserted at the beginning of the function
Function* f = ...;
IRBuilder<> entryBuilder(&*f->getEntryBlock().getFirstInsertionPt());
FunctionCallee timeStartFn = M.getOrInsertFunction("__prof_time_start",
FunctionType::get(Type::getVoidTy(Ctx), {Type::getInt64Ty(Ctx)}, false));
entryBuilder.CreateCall(timeStartFn, {ConstantInt::get(Type::getInt64Ty(Ctx), hash(f->getName()))});

// Inserted before all return/invoke instructions
for (auto& BB : *f) {
Instruction* term = BB.getTerminator();
if (isa<ReturnInst>(term)) {
        IRBuilder<> exitBuilder(term);
        FunctionCallee timeEndFn = M.getOrInsertFunction("__prof_time_end",
        FunctionType::get(Type::getVoidTy(Ctx), {Type::getInt64Ty(Ctx)},
                        false));
        exitBuilder.CreateCall(
                timeEndFn, {ConstantInt::get(Type::getInt64Ty(Ctx),
                        hash(f->getName()))});
        }
}
```

The runtime implementation of __prof_time_start / __prof_time_end writes timestamps to a memory-mapped circular buffer (mmap) read by the agent [7]. Function calls are matched using a unique ID (hashed function name), and average latencies are computed.

Loop Instrumentation via LoopInfo. Loops are identified using LoopInfoWrapperPass. A counter and entry tracker are inserted into each loop header to later analyze nesting depth, hotness, and divergence.

```
LoopInfo& LI = getAnalysis<LoopInfoWrapperPass>(*F).getLoopInfo();
for (auto* L : LI) {
BasicBlock* header = L->getHeader();
```

```
IRBuilder<> builder(&*header->getFirstInsertionPt());
FunctionCallee iterLogFn = M.getOrInsertFunction("__prof_loop_iter",
FunctionType::get(Type::getVoidTy(Ctx), {Type::getInt64Ty(Ctx)}, false));
builder.CreateCall(iterLogFn, {ConstantInt::get(Type::getInt64Ty(Ctx),
                loop_hash(header))});
}
```

This enables profiling of not only iteration counts but also behavioral changes over time—e.g., identifying unpredictable loops, cache-intensive paths, or inner branches that warrant aggressive optimization [3], [7].

Dynamic Detection of External Calls. In functions invoking external APIs (e.g., I/O, GPU, CUDA), __prof_call_entry/__prof_call_exit handlers are inserted. These allow runtime tracking of high-cost calls via CallBase analysis:

```
for (auto& BB : *f) {
for (auto& I : BB) {
    if (auto* call = dyn_cast<CallBase>(&I)) {
        if (call->isIndirectCall() || call->getCalledFunction() == nullptr) con-
  tinue;
        StringRef calleeName = call->getCalledFunction()->getName();
        if (is_expensive_external(calleeName)) {
            IRBuilder<> builder(call);
            FunctionCallee entryFn =
  M.getOrInsertFunction("__prof_call_entry",
            FunctionType::get(Type::getVoidTy(Ctx),
  {Type::getInt64Ty(Ctx)}, false));
            builder.CreateCall(entryFn, {Con-
  stantInt::get(Type::getInt64Ty(Ctx), hash(calleeName))});
        }
    }
}
}
```

The runtime agent maintains an external_call_registry with statistics (latency, frequency, standard deviation) for each external call type [2], [7]. This data is used to generate heatmaps and locate bottlenecks in dependency chains.

Memory Allocation Logging. When malloc or other allocators are detected, instrumentation injects a wrapper that logs allocation size:

```
IRBuilder<> builder(callInst);
LLVMContext& Ctx = M.getContext();
FunctionCallee logAlloc = M.getOrInsertFunction("log_alloc", Function-
Type::get(Type::getVoidTy(Ctx), {Type::getInt64Ty(Ctx)}, false));
Value* allocSize = callInst->getArgOperand(0);
builder.CreateCall(logAlloc, {allocSize});
```


ISSN 1562-9945 (Print)
ISSN 2707-7977 (Online)

This allows the profiler to track memory usage patterns with byte-level precision, which is vital in resource-constrained systems like embedded or mobile platforms.

Conditional Branch Logging. For branches with frequent divergence, the execution path is logged to detect prediction failures:

```
BranchInst* br = dyn_cast<BranchInst>(termInst);
if (br && br->isConditional()) {
IRBuilder<> builder(br);
Value* condition = br->getCondition();
Value* conditionInt = builder.CreateZExt(condition, Type::getInt32Ty(Ctx));
FunctionCallee logBranch = M.getOrInsertFunction("log_branch",
    FunctionType::get(Type::getVoidTy(Ctx), {Type::getInt32Ty(Ctx)}, false));
builder.CreateCall(logBranch, {conditionInt});
}
```

Such instrumentation helps identify irregular branch behavior, guiding transformations like reordering conditions or loop fusion.

All examples are implemented within a single LLVM ModulePass, executed early in the IR construction phase. The resulting metrics provide high-fidelity execution profiles used by the re-JIT engine for real-time optimization.

Metrics. A core component of the compiler's operation is the construction of a high-quality runtime profile based on metrics collected during execution. At each stage of analysis, key execution characteristics are gathered, including:

- function call frequency
- average and maximum basic block execution time
- per-function memory allocations
- cache miss frequency
- instruction counts within IR regions

Special attention is paid to inefficiency indicators such as branch divergence, long data dependency chains, and repeated cache line usage.

Metric collection is done through two parallel channels:

1. Instrumentation-based logging, as described in Section 4.2, captures semantic program behavior using injected IR hooks.

2. Hardware counters, such as those accessed via eBPF or PAPI, measure low-level architectural events.

For systems without access to performance monitoring units (PMUs), the compiler uses std::chrono::high_resolution_clock to record precise timing intervals between events.

All gathered metrics are aggregated into per-function profile records, updated asynchronously by a background analysis thread. Once a function crosses a defined threshold in call count or execution time, it is queued for recompilation.

This approach allows the compiler to focus resources on performance-critical regions while progressively adapting the IR to runtime conditions without developer intervention.

Testing Conditions. To evaluate the effectiveness of the proposed self-profiling architecture, a series of experiments were conducted on three types of hardware platforms, cover-

ing a wide range of applications — from desktop and server-grade processors to energy-efficient embedded systems.

Platform 1: Intel Core i7-12700H — a laptop processor with 14 physical cores (6 performance + 8 efficient), 20 threads total, max clock of 4.7 GHz, and 24 MB of L3 cache. This setup simulates high-throughput desktop workloads that are sensitive to dynamic load.

Platform 2: AMD Ryzen 5 3600 — a six-core desktop CPU with SMT (12 threads), 3.6 GHz base frequency, and high memory bandwidth. This represents typical multithreaded compilation tasks.

Platform 3: Raspberry Pi 4B — an ARM Cortex-A72-based SBC (4 cores @ 1.5 GHz) with 4 GB RAM. This configuration was chosen to evaluate self-profiling viability on constrained devices [7].

On all platforms, the same benchmark application was used: a medium-intensity numerical workload involving internal loops, external API calls, randomized memory access, and dynamic branch structure. Code was compiled in two modes:

1. With self-profiling enabled
2. Without self-profiling (baseline)

Each configuration was executed 20 times, with cache flushing and warm-up phases to ensure consistency.

Measured Metrics. During each benchmark execution, three primary metrics were recorded:

- Average program performance, expressed as frames per second (FPS) or basic blocks processed per second depending on the mode.
- Total compilation time, including time spent on dynamic re-compilation of hot regions.
- Relative performance gain compared to the baseline (no self-profiling).

Intel Core i7-12700H showed a performance increase of approximately 15%. The baseline ran at 112 FPS, while the self-profiling configuration consistently reached 129 FPS. The compilation time was about 18 ms longer on average but was offset by improved optimization decisions. In 17 out of 20 runs, performance gains were observed.

AMD Ryzen 5 3600 demonstrated a gain of about 11–13%. While fewer cores and less aggressive thread scheduling led to slightly lower peak throughput, self-profiling still reduced the number of suboptimal transformations. Re-compilation accounted for no more than 3% of total runtime.

Raspberry Pi 4B, despite its limited compute resources, yielded an approximate 9% improvement. The profiler successfully identified and optimized inefficient regions—especially those related to branching and cache pressure. No manual tuning or environmental adjustments were necessary; the system operated autonomously with minimal overhead.

Experimental Conclusions. Overall, the experimental results confirm that integrating self-profiling into the compiler pipeline yields measurable performance benefits without compromising code stability or compatibility.

ISSN 1562-9945 (Print)
ISSN 2707-7977 (Online)

In 87% of test runs, re-compilation of hot regions based on runtime metrics led to more effective combinations of optimization passes. These improved configurations, in turn, reduced the total execution time of the application [1].

The time spent collecting profile data and performing selective re-compilation was minimal compared to the performance gains—especially in scenarios involving long-running or high-load applications. In some cases, such as data streaming or sensor-based processing, the observed speedups reached 20–25%.

These results demonstrate the practicality of using real-time feedback to guide compilation strategies dynamically. The system proves effective not only on powerful desktop platforms but also on constrained environments [7], [10], highlighting its general applicability to compilers of the next generation.

Practical Significance of Results. The practical value of this research lies in its ability to significantly improve runtime performance without requiring additional developer effort. This is especially beneficial for high-performance computing (HPC), edge analytics, mobile applications, and other resource-sensitive domains.

The greatest benefits of self-profiling were observed in scenarios involving:

- significant workload variability,
- frequent calls to performance-critical functions.

In contrast, applications with stable and predictable execution behavior exhibited smaller relative improvements. However, even in those cases, the system imposed no meaningful overhead and maintained overall stability.

By reducing the need for manual tuning and increasing execution efficiency, self-profiling compilers can simplify development workflows while enabling real-time performance adaptability. This is particularly valuable in environments where deployment contexts change dynamically or cannot be predicted in advance.

Conclusions and Future Work. This paper introduced a compiler architecture that supports real-time self-profiling and enables adaptive code translation based on dynamic execution characteristics. The proposed approach integrates LLVM's traditional toolchain with high-resolution runtime monitoring, dynamic performance analysis, and selective IR transformations without stopping or restarting the application.

The system empowers the compiler to detect so-called "hot" code regions—those with the greatest impact on performance—and adjust optimization profiles accordingly. This transforms the compiler into an active performance agent capable of adapting to runtime context, hardware configuration, and input structure without developer intervention. Such capabilities are particularly important in heterogeneous systems that combine diverse compute resources and exhibit variable workload patterns.

Experiments showed that re-JITing hot functions based on runtime feedback improved performance by 10–20% on average, with no loss of stability or significant overhead. Unlike traditional Profile-Guided Optimization (PGO), which requires a separate run for data collection and recompilation, the proposed system performs all stages during a single execution cycle. This makes it suitable for use cases where agility and responsiveness are critical—such as

stream processing, real-time control, edge analytics, sensor pipelines, and high-performance cluster workloads.

Our analysis suggests that integrating self-profiling into modern compilers is not only technically feasible but strategically sound. It shifts the compilation paradigm from static to dynamic, paving the way for intelligent compilers that learn from application behavior rather than relying solely on heuristics or predefined flag profiles.

Future research will explore several directions:

- Hybrid target support, where compilation decisions include both optimization passes and compute platform selection (e.g., offloading to GPU/TPU).
- Deeper LLVM ORC JIT v2 integration, enabling multi-tier JIT, persistent caching, and condition-based recompilation.
- Machine learning integration, especially reinforcement learning, for automatic selection of optimization sequences based on performance metrics.
- Adaptation to constrained platforms, such as WebAssembly and ARM-based mobile devices, which requires minimizing agent size and further optimizing metric collection under strict energy and latency constraints.

Taken together, these directions will strengthen the role of self-profiling as a foundational component of future compiler ecosystems - flexible, adaptive, and self-improving.

REFERENCES

1. Muchnick, S. S. (1997). Advanced Compiler Design and Implementation. Morgan Kaufmann.
2. Cooper, K. D., & Torczon, L. (2011). Engineering a Compiler (2nd ed.). Morgan Kaufmann.
3. Aycock, J. (2003). A Brief History of Just-In-Time Compilation. Springer.
4. Tratt, L. (2021). Modern Compiler Implementation in a Post-LLVM World. Springer.
5. Parr, T. J. (2010). The Definitive ANTLR 4 Reference: Building Domain-Specific Languages. Pragmatic Bookshelf.
6. Lattner, C., & Adve, V. (2004). LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. University of Illinois.
7. Jones, R., Hosking, A., & Moss, E. (2011). The Garbage Collection Handbook: The Art of Automatic Memory Management. Chapman & Hall/CRC.
8. Nystrom, R. (2021). Crafting Interpreters. Genever Benning.
9. Aho, A. V., Lam, M. S., Sethi, R., & Ullman, J. D. (2006). Compilers: Principles, Techniques, and Tools (2nd ed.). Addison-Wesley.
10. Appel, A. W. (1998). Modern Compiler Implementation in C. Cambridge University Press.

Received 20.08.2025.
Accepted 27.08.2025.

ISSN 1562-9945 (Print)
ISSN 2707-7977 (Online)

Механізми самопрофілювання компіляторів для коду в реальному часі

У статті досліджується концепція самопрофілювальних компіляторів як засобу адаптивної оптимізації коду в реальному часі. Підхід ґрунтується на збиранні динамічних метрик продуктивності під час виконання програми та аналізі отриманих даних для вибору найефективніших стратегій компіляції. Запропоновано архітектуру компілятора, здатного автоматично виявляти критичні з точки зору продуктивності ділянки коду (hotspots), налаштовувати конфігурацію оптимізаційних проходів і здійснювати повторну компіляцію з урахуванням оновлених метрик. Реалізовано прототип на основі LLVM з вбудованим агентом часу виконання, відповідальним за інструментування коду, збір метрик та взаємодію з динамічним Pass Manager. Проведено серію експериментів на різних апаратних платформах, зокрема на десктопних процесорах і ARM-архітектурах. Результати продемонстрували суттєве зростання продуктивності без помітного збільшення часу компіляції або використання ресурсів. Отримані дані підтверджують доцільність інтеграції самопрофілювання в компілятори наступного покоління, орієнтовані на високопродуктивні обчислення, edge-системи та мобільні пристрої. У роботі представлено концепцію самопрофілювання як інструмент реальної оптимізації коду. Реалізовано прототип на базі LLVM із вбудованим модулем runtime-аналізу. Результати демонструють переваги запропонованого підходу.

Ключові слова: адаптивна компіляція; оптимізація часу виконання; самопрофілювання; LLVM; JIT-компіляція; продуктивність.

Бердник Михайло – доктор технічних наук, доцент, професор кафедри програмного забезпечення комп'ютерних систем, Національний технічний університет «Дніпровська політехніка», м. Дніпро, Україна, ORCID: https://orcid.org/0000-0003-4894-8995
Стародубський Ігор – аспірант кафедри програмного забезпечення комп'ютерних систем, Національний технічний університет «Дніпровська політехніка», м. Дніпро, Україна, ORCID: https://orcid.org/0009-0004-1864-7889

Berdnyk Mykhailo – Doctor of Technical Sciences, Associate Professor, Professor of the Department of Software for Computer Systems, Dnipro University of Technology, Dnipro, Ukraine, ORCID: https://orcid.org/0000-0003-4894-8995
Starodubskyi Igor – PhD Student of the Department of Software for Computer Systems, Dnipro University of Technology, Dnipro, Ukraine, ORCID: https://orcid.org/0009-0004-1864-7889