Микроархитектурно-зависимая оптимизация вычислений в интеллектуальных системах

А. А. Когутенко

Санкт-Петербургский государственный электротехнический университет «ЛЭТИ» им. В.И. Ульянова (Ленина)

and.kogutenko@yandex.ru

***Аннотация*. В работе рассматриваются ограничения статических компиляторов и средств профилирования (PGO, BOLT) при оптимизации вычислительных циклов на гетерогенных процессорах. Выявлено, что статический анализ не способен эффективно предсказывать аномалии производительности, вызванные микроархитектурными особенностями. Предложена концепция легковесного адаптивного JIT-оптимизатора, основанного на апробации эквивалентных вариаций машинного кода во время работы программы. Рассмотрены перспективы данного подхода для нивелирования аппаратной фрагментации архитектуры RISC-V.**

Ключевые слова: микроархитектура; динамическая оптимизация; JIT-компиляция; профильная оптимизация; гетерогенные процессоры; x86-64; RISC-V

I. Введение

В современных интеллектуальных системах основное процессорное время затрачивается на выполнение вычислительно нагруженных циклов (тензорные операции, фильтрация, инференс нейронных сетей). Для наиболее эффективного использования доступных ресурсов центрального процессора (CPU) требуется задействование сложных низкоуровневых оптимизаций машинного кода.

Традиционный подход, в основе которого лежит предварительная статическая компиляция (AOT), генерирует код, оптимизированный под усреднённую микроархитектуру. Как показывает практика, даже использование профильной оптимизации (Profile-Guided Optimization, PGO) и компиляция с указанием целевого процессора не гарантируют достижения максимальной производительности, особенно в случае использования гетерогенных процессоров (например, имеющих производительные и энергоэффективные ядра).

Это связано с тем, что статический анализ принципиально не способен учесть скрытое состояние процессора в момент выполнения: влияние точного выравнивания инструкций в кэш-памяти, паттерны слияния операций (micro и macro fusion [1]) или разницу в стоимости сброса конвейера для разных типов ядер. Попытки нивелировать эти факторы с помощью утилит пост-линковочной оптимизации бинарного кода (таких как LLVM BOLT [2]) также опираются на заранее собранную статистику и лишены гибкости. Более того, взаимодействие кода с аппаратными механизмами (например, блоками выборки и декодирования команд) может приводить к парадоксальным эффектам, когда применение стандартных методов оптимизации, таких как выравнивание инструкций по границе кэш-линии, приводит к деградации производительности вместо ожидаемого прироста. В связи с этим возникает объективная необходимость в разработке методов адаптации машинного кода непосредственно во время выполнения (JIT-трансформации).

Цель данной работы – исследовать эффективность применения низкоуровневых оптимизаций различными компиляторами на современных микроархитектурах, а также разработать концепцию оптимизатора, способного перестраивать машинный код высоконагруженных циклов «на лету» без существенных накладных расходов, присущих традиционным средам трансляции программ в реальном времени. Отдельное внимание в статье уделено перспективам применения предложенных идей для активно развивающейся архитектуры RISC-V в условиях её аппаратной фрагментации.

II. Методология тестирования

Для выявления проблем и ограничений статической кодогенерации было осуществлено тестирование на широком спектре микроархитектур x86-64. Основной акцент сделан на процессоры с гетерогенной архитектурой (в частности, Intel Core i7-13700H, объединяющий производительные P-ядра Raptor Cove и энергоэффективные E-ядра Gracemont), а также на микроархитектуры Intel Gemini Lake (Celeron J4105) и AMD Zen 3 (Ryzen 5 5600H).

Генерация машинного кода выполнялась при помощи компиляторов, широко применяемых в промышленной разработке: GCC 13.1.1, Clang 15.0.6 и Intel C Compiler (ICC) 2023.1.0. Для оценки эффективности современных алгоритмов кодогенерации применялись технология профильной оптимизации PGO и утилита пост-линковочной оптимизации бинарного кода LLVM BOLT.

С целью обеспечения высокой воспроизводимости результатов и минимизации аппаратных погрешностей, вносимых технологиями динамического изменения тактовой частоты (Turbo Boost) и влиянием работы планировщика операционной системы, многократные прогоны бенчмарков осуществлялись на фиксированной частоте процессора с активацией профиля максимальной производительности.

Для полноты исследования и сопоставления предлагаемого подхода с существующими методами динамической трансформации, в работе также уделено внимание возможностям инструментов динамической инструментации (таких как QEMU [3] и DynamoRIO [4]). Это позволяет оценить проблему накладных расходов, которые являются критическим барьером для внедрения методов динамической оптимизации в системы, требовательные к производительности.

III. Результаты и анализ проблем статической кодогенерации

Современные процессоры задействуют множество сложнейших технологий, среди которых внеочередное и спекулятивное исполнение команд, суперскалярность, многоуровневая иерархия кэш-памяти и продвинутые механизмы предсказания ветвлений. Это приводит к тому, что стандартные бенчмарки часто оказываются недостаточно чувствительными к микроархитектурным особенностям.

Для комплексной оценки качества низкоуровневых оптимизаций был разработан ряд синтетических тестов, направленных на анализ эффективности кодогенератора компиляторов и выявление неочевидных особенностей микроархитектуры процессоров. Основное внимание уделено следующим аспектам:

- оценке влияния выравнивания кода на качество работы блоков выборки и декодирования команд;
- эффективности задействования компиляторами инструкций условного копирования (CMOV) для минимизации сбросов конвейера;
- анализу качества автоматической векторизации.

A. *Выравнивание нагруженных циклов*

Одним из факторов для достижения максимальной производительности нагруженных циклов является их выравнивание по границе кэш-линии (64 байта на системах x86-64). Для проверки этого факта была написана следующая программа на языке Ассемблера:

```
mov rcx, ITERATIONS
nop
...
nop
loop_begin:
    nop
    loop loop_begin
```

Запуск 64-х вариаций этой программы, отличающихся только числом NOP-инструкций перед началом нагруженного цикла, на P-ядрах процессора Intel Core i7-13700H (микроархитектура Raptor Cove) завершился предсказуемо – в зависимости от удачности выравнивания программа работала быстрее (порядка 1 с) или медленнее (почти 2 с).

Однако, запуск этих же вариаций программы на E-ядрах того же процессора (Gracemont) показывает отсутствие влияния выравнивания. Это означает, что в некоторых случаях применение подобной оптимизации не является целесообразным, и может негативным образом повлиять на эффективность использования кэш-памяти команд.

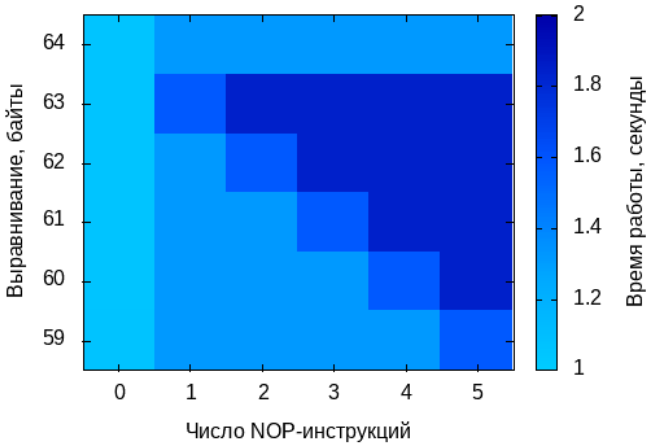

Рис. 1. Результаты запуска на P-ядре Intel Core i7-13700H

B. *Минимизация штрафов за ветвления и пределы эффективности PGO/BOLT*

На задаче с множеством условных переходов (проверка гипотезы Коллатца) оценивалась базовая компиляция, а также эффективность применения PGO и утилиты BOLT. Установлено, что компилятор GCC (время выполнения 2,36 с на P-ядрах Raptor Cove) критически уступает аналогам Clang на базе инфраструктуры LLVM (1,71 с) и ICC (1,56 с). Причина кроется в отказе эвристик GCC от использования инструкции условного копирования CMOV, предотвращающей сброс конвейера при неудачном предсказании ветвления.

Более того, применение пост-линковочной раскладки кода с помощью LLVM BOLT не всегда оказывало положительное влияние. Как отмечается в современных индустриальных исследованиях [5], выигрыш от BOLT часто ограничен единицами процентов, однако в нашем случае на архитектуре Gemini Lake применение утилиты к коду GCC привело к явной деградации (увеличение времени выполнения с 5,98 с до 6,48 с). Это доказывает, что статические профили не могут универсально улучшить код для всех микроархитектурных реализаций.

C. *Проблема автовекторизации и пределы скалярной оптимизации компиляторов*

Критической проблемой оказалась ловушка неконтролируемой автовекторизации. При тестировании алгоритма сортировки пузырьком на P-ядрах i7-13700H код GCC (с флагом *-O2*) выполнялся за 6,00 с. Принудительное отключение векторизации флагом *-fno-tree-vectorize* ускорило работу программы почти в два раза – до 3,13 с. Стоит отметить, что профильная оптимизация (PGO) в связке с Clang (с опциями *-Ofast -march=native*) на E-ядрах Gracemont также привела к сильной регрессии производительности (увеличение времени с 2,53 с до 3,51 с). Агрессивные векторные оптимизации компилятора, применяемые без учёта специфики целевого устройства, способны разрушить скалярную производительность.

ТАБЛИЦА I. РЕЗУЛЬТАТЫ ТЕСТИРОВАНИЯ ГИПОТЕЗЫ КОЛЛАТЦА ПРИ РАЗЛИЧНЫХ ОПЦИЯХ КОМПИЛЯЦИИ

	13700H P-Core 3.8 GHz (Raptor Cove)		**13700H E-Core 3.1 GHz (Gracemont)**		**J4105 1.5 GHz (Gemini Lake)**		**5600H 3.3 GHz (Zen 3)**	
	Базовый	BOLT	Базовый	BOLT	Базовый	BOLT	Базовый	BOLT
GCC O2	2.364	2.261	2.793	3.091	5.987	6.483	2.254	2.254
GCC O2 + PGO	2.123	2.123	2.687	3.090	5.975	6.494	2.058	2.057
GCC Ofast march=native	2.344	2.267	2.794	3.090	5.964	6.492	1.983	1.985
GCC Ofast march=native + PGO	2.123	2.255	2.687	3.090	5.970	6.491	1.984	1.983
Clang O2	1.710	1.680	2.411	2.388	5.758	5.722	1.482	1.482
Clang O2 + PGO	1.499	1.436	2.520	2.517	6.109	6.121	1.492	1.491
Clang Ofast march=native	1.715	1.685	2.193	2.178	5.393	5.455	1.484	1.484
Clang Ofast march=native + PGO	1.726	1.743	2.161	2.163	5.405	5.405	1.491	1.493
ICC O2	1.568	1.624	2.181	2.182	5.059	5.051	—	—
ICC O2 + PGO	1.634	1.572	2.175	2.179	5.079	5.077	—	—
ICC Ofast march=native	1.566	1.584	2.073	2.076	4.806	4.792	—	—
ICC Ofast march=native + PGO	1.618	1.602	2.072	2.070	4.784	4.790	—	—

Ярким подтверждением ограниченности статических эвристик стало сравнение скомпилированного кода с реализацией, написанной вручную на языке ассемблера (NASM). Ручная ассемблерная версия алгоритма сортировки пузырьком, использующая CMOV и оптимизированное распределение регистров, оказалась быстрее созданного компилятором кода практически на всех протестированных микроархитектурах.

Наиболее впечатляющий результат был зафиксирован на энергоэффективных E-ядрах Gracemont (i7-13700H): время выполнения ручного ассемблерного кода составило 1,22 с, что более чем на 70% быстрее лучшего результата компилятора Clang с агрессивным флагом оптимизации *-Ofast* (2,87 с). Использование PGO-профилирования для данного алгоритма не привело к улучшению результатов. Это доказывает, что в условиях сложного графа потока управления компиляторы до сих пор не способны генерировать идеальный скалярный машинный код, проигрывая человеку и оставляя пространство для работы JIT-супероптимизатора.

D. Аномалия «лишней» инструкции

Невозможность создания идеальной статической эвристики для генерации машинного кода ярко иллюстрируется обнаруженным микроархитектурным парадоксом. В ходе тестирования короткого цикла задержки, выровненного по границе 64 байт (размер кэш-линии L1 инструкций на x86-64 системах) и состоящего только из декремента и условного перехода:

```
mov rcx, ITERATIONS
loop_begin:
    dec rcx
    jnz loop_begin
```

на P-ядрах процессора i7-13700H было зафиксировано стабильное время выполнения 1,6 с. Однако искусственное добавление функционально бесполезной инструкции NOP внутрь тела этого цикла:

```
mov rcx, ITERATIONS
loop_begin:
    dec rcx
    nop
    jnz loop_begin
```

неожиданным образом привело к ускорению времени выполнения в случайном интервале от 0,95 с до 1,25 с. Значительный разброс времени и сам факт ускорения (при зафиксированной тактовой частоте) указывают на сложную внутреннюю логику работы фронтенда процессора. Добавление NOP смещает инструкции внутри окна выборки, что непредсказуемо влияет на кэш микроопераций (uop-cache), Loop Stream Detector (LSD) и паттерны макро-слияния (macro-op fusion) [1].

IV. КОНЦЕПЦИЯ АДАПТИВНОГО ДИНАМИЧЕСКОГО ОПТИМИЗАТОРА

Фундаментальная возможность ускорения статически скомпилированного машинного кода во время выполнения была доказана ещё в классическом проекте HP Dynamo [6]. Однако оригинальный подход Dynamo и его современные аналоги (такие как фреймворки динамической инструментации QEMU и DynamoRIO) опираются на интерпретацию и непрерывное отслеживание всего графа потока управления для поиска «горячих трасс» (hot traces). Подобная архитектура вносит недопустимо высокие накладные расходы.

Актуальность создания средств JIT-оптимизации подтверждается последними индустриальными тенденциями. Весной 2026 года компания Intel представила технологию Binary Optimization Tool (BOT) [7]. Данный инструмент посредством фоновой службы отслеживает запуск поддерживаемых приложений и «на лету» перенаправляет поток управления на заранее оптимизированные (например, векторизованные) части кода, оставляя оригинальный бинарный файл на диске неизменным. Однако Intel BOT является проприетарным решением, которое функционирует исключительно на базе готовых профилей, поставляемых производителем для ограниченного набора программ.

В отличие от закрытых инструментов, полагающихся на заранее подготовленные шаблоны, предлагаемая концепция адаптивного JIT-оптимизатора является полностью автономной. Для преодоления ограничений статического анализа и устранения высоких накладных расходов традиционных сред трансляции, предлагается создать легковесный инструмент, самостоятельно генерирующий и анализирующий качество кодовых трансформаций прямо «на лету», адаптируясь под конкретный кристалл.

Эффективная реализация подобного инструмента подразумевает несколько аспектов:

1. Изоляция точек вмешательства. В отличие от классических DBI-инструментов, точки возврата управления в среду трансформации размещаются исключительно вне вычислительно нагруженных циклов (до и после них), что гарантирует отсутствие накладных расходов внутри критических секций.
2. Апробация трансформаций (микротестирование). Так как микроархитектурные эффекты непредсказуемы (что подтверждается аномалией с инструкцией NOP), среда генерирует несколько семантически эквивалентных вариантов тела цикла. Каждый вариант выполняется на малом числе итераций, после чего измеряется затраченное время. Наиболее оптимальный вариант кэшируется и используется для обработки оставшегося массива данных.
3. Предварительный просчёт и JIT-супероптимизация. Для минимизации задержек выполнения, оптимизатор профилирует целевую систему предварительно. Крайне перспективным видится применение методов супероптимизации. Интеграция современных решений, таких как SIMD-ориентированный оптимизатор Minotaur [8] или ИИ-фреймворк Iago [9], синтезирующий нетривиальные графы инструкций, позволит находить неочевидные микроархитектурные решения, принципиально недоступные классическим AOT-компиляторам.

V. Перспективы для архитектуры RISC-V

Предложенный метод имеет высокий потенциал применения на открытой архитектуре RISC-V, экосистема которой характеризуется глубокой аппаратной фрагментацией. В базовой спецификации RISC-V отсутствует инструкция условного копирования. Для сокращения переходов разработано расширение Zicond [10]. Однако, как отмечают архитекторы процессоров, применение данного расширения не даёт существенного прироста производительности без наличия в кристалле сложного механизма макро-слияния инструкций. Статическому компилятору эта информация недоступна. Динамический оптимизатор способен «на лету» протестировать классическое ветвление и Zicond-последовательность, выбрав оптимальную для конкретного SoC. Кроме того, векторное расширение RISC-V (RVV) является архитектурно-независимым, и длина векторного регистра (VLEN) неизвестна компилятору. Применение адаптивного JIT-подхода позволит динамически вычислять идеальный шаг разворачивания циклов для любого целевого нейросетевого процессора без перекомпиляции приложения.

Стоит отметить, что открытость архитектуры RISC-V позволяет дополнять уже существующую экосистему собственными расширениями. Это даёт возможность улучшить эффективность предлагаемого инструмента за счёт вспомогательных инструкций, снижающих накладные расходы на переключение контекста оптимизатора и тестируемой программы.

Например, динамический оптимизатор может часто делать агрессивные предположения – например, «в этом цикле число A всегда равно X». Если предположение оказывается неверным, процессор инициирует аппаратное микро-исключение и переводит управление на неоптимизированную статическую версию цикла.

VI. Заключение

Проведенный анализ доказал, что гетерогенность и сложность современных микроархитектур ограничивают эффективность оптимизаций AOT-компиляторов и утилит, ориентированных на использование профиля. Неспособность статического анализа предсказывать аппаратные эффекты делает необходимым переход к адаптивной кодогенерации во время выполнения. Предложенная концепция легковесного оптимизатора с динамическим тестированием является эффективным решением проблемы фрагментации аппаратного обеспечения как для архитектуры x86-64, так и для активно развивающейся экосистемы RISC-V.

Список литературы

[1] Fog A. The microarchitecture of Intel, AMD and VIA CPUs: An optimization guide for assembly programmers and compiler makers. Copenhagen University College of Engineering, 2024. 277 p.

[2] Panchenko M., Auler R., Nell B., Ottoni G. BOLT: a practical binary optimizer for data centers and beyond. Proceedings of the 2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO), 2019, pp. 2-14.

[3] Bellard F. QEMU, a fast and portable dynamic translator. USENIX Annual Technical Conference, FREENIX Track, 2005, vol. 41, pp. 46-50.

[4] Bruening, D. An Infrastructure for Adaptive Dynamic Optimization / D. Bruening, T. Garnett, S. Amarasinghe // Proceedings of the International Symposium on Code Generation and Optimization. San Francisco, 2003. P. 265–275.

[5] Liu B. BOLT Optimization Technology Could Bring Obvious Performance Uplift On Arm Server. SemiEngineering Technology, 2024.

[6] Bala V., Duesterwald E., Banerjia S. Dynamo: a transparent dynamic optimization system. Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation (PLDI '00), 2000, pp. 1-12.

[7] Intel's Binary Optimization Tool tested and explained. Tom's Hardware, 2026. Доступно на: https://www.tomshardware.com (дата обращения: 30 марта 2026).

[8] Regehr J., Liu Z. Minotaur: A SIMD-Oriented Synthesizing Superoptimizer. 3rd Workshop on LLVM in Parallel Processing (LLPP), 2023.

[9] Mukherjee M. Iago: AI Driven Superoptimization for LLVM. Proceedings of the 2025 US LLVM Developers' Meeting, 2025.

[10] Hower D. RISC-V Zicond Extension Evaluation and Microarchitectural Impact. RISC-V International Architecture Review, 2023. Доступно на: https://github.com/riscv/riscv-zicond (дата обращения 30 марта 2026).