Санкт-Петербургский государственный университет

Кафедра системного программирования

Группа 22.Б09-мм

Анализ производительности приложения k64

Десятников Павел Викторович

Отчёт по учебной практике
в форме «Решение»

Научный руководитель:
ст. преп. каф. ИАС, К. К. Смирнов

Консультант:
старший разработчик ПО I категории, ООО «Софтком», П. А. Бабанов

Санкт-Петербург
2024

Оглавление

Введение

В современном мире эмуляторы процессоров являются важной частью разработки и тестирования программного обеспечения, так как их применение имеет ряд преимуществ перед использованием реального аппаратного обеспечения. К примеру, эмуляторы позволяют разрабатывать ПО для вычислителя, осязаемого экземпляра которого ещё не существует, а также они устраняют проблему неудобных отладочных интерфейсов целевых вычислителей [2]. Существует множество решений с различным уровнем сложности и количеством доступных архитектур.

Как правило, время работы программы на эмуляторе больше, чем на реальной машине, поэтому задача анализа его производительности и последующей оптимизации остаётся актуальной.

Объектом изучения в данной работе является эмулятор процессора КОМДИВ-64 [8]. Данный процессор реализует набор команд архитектуры MIPS64, а эмулятор был разработан в ООО «Софтком» и получил название k64. Одной из его ключевых особенностей является то, что написан он в основном на ассемблере. Это сильно усложняет задачу анализа его производительности, поскольку дополнительную информацию для успешного профилирования придётся проставлять вручную. Более того, в интересах компании требуется перенести проект под Linux, так как на данный момент эмулятор работает только на платформе Windows.

1. Постановка задачи

Целью работы является изучение производительности приложения k64.

Для её выполнения были поставлены следующие задачи:

1. перенести приложение под Linux, сохранив возможность сборки под Windows;
2. расставить DWARF метки для профилирования;
3. проанализировать производительность приложения k64.

2. Обзор

2.1. Инструменты анализа производительности

Существует множество инструментов анализа производительности приложений [10]. Упомянем наиболее известные из них: perf [11], gprof [6], Valgrind Callgrind [13]. Также стоит упомянуть Intel Vtune [7] и AMD uProf [1].

Perf — мощный инструмент, являющийся частью ядра Linux. Он предназначен для разностороннего анализа производительности как на аппаратном, так и на программном уровне.

Gprof — инструмент анализа производительности для приложений Unix. Использует технологию, соединяющую в себе сэмплирование[1] и инструментирование[2].

Valgrind — фреймворк для создания инструментов разностороннего анализа приложений. Одним из таких инструментов является Callgrind, способный строить дерево стека вызовов.

Intel Vtune — анализатор производительности приложений, работающий на процессорах от Intel. Использует различные технологии профилирования, в том числе обладает необходимым инструментарием для анализа многопоточных программ. Доступен как на Windows, так и на Linux.

AMD uProf — инструмент, схожий по области применения с Intel Vtune, но для процессоров от AMD.

Для дальнейшей работы было принято решение использовать perf, поскольку в таком случае для визуализации полученных данных удобно воспользоваться проектом FlameGraph [5].

[1]Сбор данных из определённого подмножества событий, происходящий раз в фиксированную единицу времени.

[2]Профилирование с вставкой дополнительного кода, генерируемого компилятором по запросу пользователя.

2.2. k64

k64 — эмулятор процессора КОМДИВ-64. Данный эмулятор был разработан в ООО «Софтком». Проект написан на языках Си и Ассемблер и предназначен для работы под Linux и Windows, но по историческим причинам работает только на платформе Windows.

3. Реализация

3.1. Перенос проекта на Linux

Изначально, в предоставленной версии проекта k64 был определён авторский макрос WINMIPS. Разработчик определял его единицей, если сборка в данный момент на Windows, и нулём в противном случае.

Сам макрос использовался для подключения правильных библиотек и функций (select, dlopen или LoadLibrary, dlsym или GetProcAddress и т. д.), а также, для соблюдения ABI (т. е. в каком порядке и в какие регистры помещать аргументы функций).

Такой подход являлся неудобным, так как приходилось при каждом переключении между операционными системами вручную переопределять макрос. Более того, сделать это было необходимо в разных местах проекта, друг с другом несогласованных, а также требовалось вносить изменения в файл CMakeLists.txt. Помимо вышеупомянутого, в процессе разработки k64 и вовсе утратил возможность корректно собираться и работать на Linux.

Для решения данной проблемы было принято решение использовать более традиционный и надёжный способ — макросы _WIN32 и __linux__ в файлах проекта, написанных на языках Си и Ассемблер, которые определены лишь в операционных системах, названиям которых они отвечают, а также WIN32 и UNIX в CMakeLists.txt.

В результате проделанных изменений, проект вернул возможность работать на Linux, при этом больше не нужно редактировать код, чтобы собирать приложение на разных ОС.

3.2. Расставление DWARF меток

Профилирование приложения сильно затруднено тем, что бóльшая часть проекта написана на ассемблере. Если бы все функции имели стандартный пролог, указанный ниже, то разворачивание стека было бы тривиальной задачей, поскольку регистр RBP в таком случае неизменен, а именно он указывает на фрейм стека (т. е. в вызываемой функ-

ции всегда хранится адрес вызывающей).

Листинг 1: Стандартный пролог функции.

```
    push RBP
    mov RBP, RSP
```

Но в нашем случае подобное соглашение не соблюдено. И если компилятор, который не добавил данный пролог в целях оптимизации, сгенерирует необходимую DWARF информацию автоматически по запросу разработчика, то в коде, написанном на ассемблере, таковую придётся расставлять вручную [3].

Но и эта задача не является тривиальной. В целях максимального увеличения производительности код проекта написан неконвенционально: например, зачастую функции не вызываются напрямую, а в них совершается прыжок, что нарушает соглашения вызовов в x86_64 ABI [12]. Например, в листинге 2 показано, что в обработчик «передаёт управление» на функцию dg_exec_exit, то есть совершает в неё прыжок.

Листинг 2: Функция из проекта k64, нарушающая соглашения вызовов.

```
exec_and:
    mov (RDI), RAX
    and (R8), RAX
    mov RAX, (RSI)
    jmp dg_exec_exit
```

В листинге 3 же представлена функция, сгенерированная компилятором gcc, расставление Call Frame Information (CFI) меток в ней, согласно руководству [4], не вызвало бы затруднений.

Листинг 3: Функция, сгенерированная компилятором gcc.

```
Double:
    push RBP
    mov RSP, RBP
    mov RDI, -0x4(RBP)
    mov -0x4(RBP), RAX
```

```
    add RAX, RAX
    pop RBP
    ret
```

В листинге 4 представлено, какой вид приняла функция из листинга 2 после расставления в ней меток.

Листинг 4: Функция из проекта k64 с проставленными CFI метками.

```
exec_and:
.cfi_startproc
.cfi_def_cfa RSP, 8
    mov (RDI), RAX
    and (R8), RAX
    mov RAX, (RSI)
    jmp dg_exec_exit
.cfi_endproc
```

Также, в одном из мест кода функция get_phis_addr, вызываемая всегда только из dg_exec, в случае обработки ошибки совершает возврат «через голову» вызывающей функции. Не удалось придумать способ успешной обработки этой ситуации.

3.3. Профилирование и генерация Flame графов

Для обработки результатов профилирования и их визуализации было принято решение использовать проект Flame Graph, разработанный Бренданом Греггом. На листинге 5 представлено, как производился сбор статистики.

Листинг 5: Сбор статистики и её обработка с построением Flame графа (все команды выполняются под рутом).

```
    perf record --call-graph dwarf -F 1000 <application>
    perf script --max-stack 4098 -i perf.data | ./FlameGraph/
stackcollapse-perf.pl > perf.data.folded
    ./FlameGraph/flamegraph.pl perf.data.folded > fg.svg
```

Схема работы довольна проста: команда perf record выполняет сэмплирование трассировки стека для указанного приложения с частотой

1000 Герц, используя DWARF информацию для разворачивания стеков, записывая всю информацию в файл perf.data. Затем perf script обрабатывает полученные трассировки стеков, stackcollapse-perf.pl преобразовывает их в линии и flamegraph.pl записывает полученные данные в файл формата SVG, который удобно просматривать через браузер.

Для тестирования эмулятора было принято использовать бенчмарк[3] под названием linpack [9].

На рисунке 1 показано, как выглядел Flame граф исходного кода проекта, до добавления в него CFI меток.

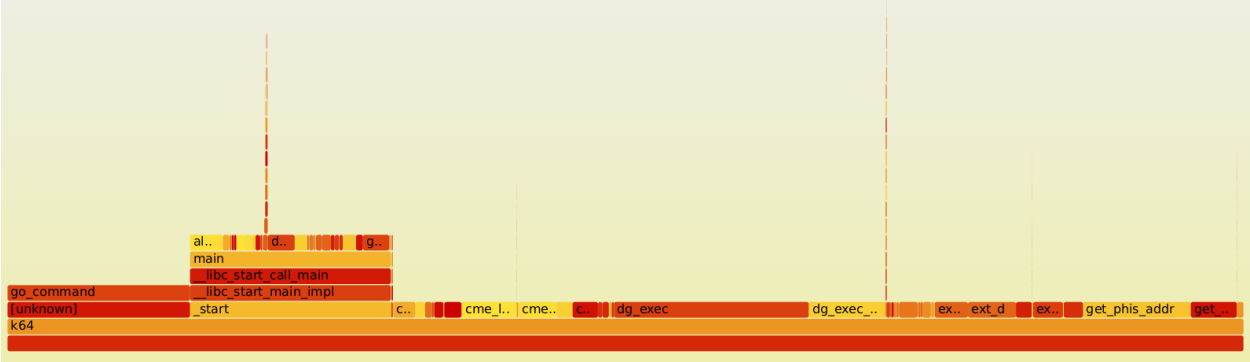


Рис. 1: Flame граф приложения k64 с запущенным бенчмарком linpack до добавления CFI меток.

На рисунке 2 представлено, как стал выглядеть Flame граф после добавления в проект CFI меток.

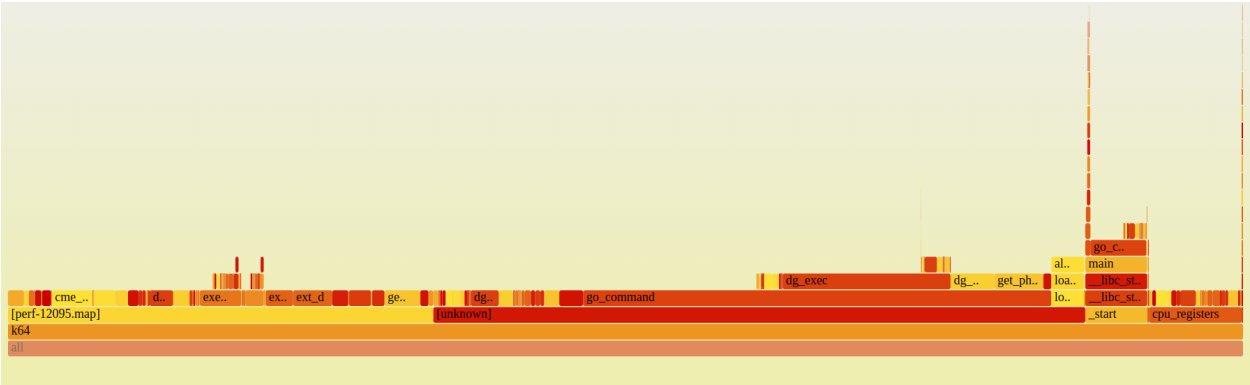


Рис. 2: Flame граф приложения k64 с запущенным бенчмарком linpack после добавления CFI меток.

[3]Программа для тестирования производительности вычислительной системы.

3.4. Анализ полученных данных

После изучения полученного Flame графа было сделано несколько выводов:

- не удалось расставить все метки полностью корректно, так как полученный график неидеален;
- эмуляция памяти является основной проблемой.

В особенности выделяется команда ldc1, исполнение которой занимает достаточно долгое время. После просмотра реализации обработчика данной инструкции стало понятно, что ускорить исполнение не получится.

С учётом предыдущего умозаключения о проблемах с эмуляцией памяти, было принято решение запустить perf с флагом для сбора данных о промахах кэша «-e cache-misses». На этот раз данные обрабатывались при помощи perf report -v. Была сгенерирована таблица, из которой видно, какой процент от всех промахов кэша приходится на конкретную функцию.

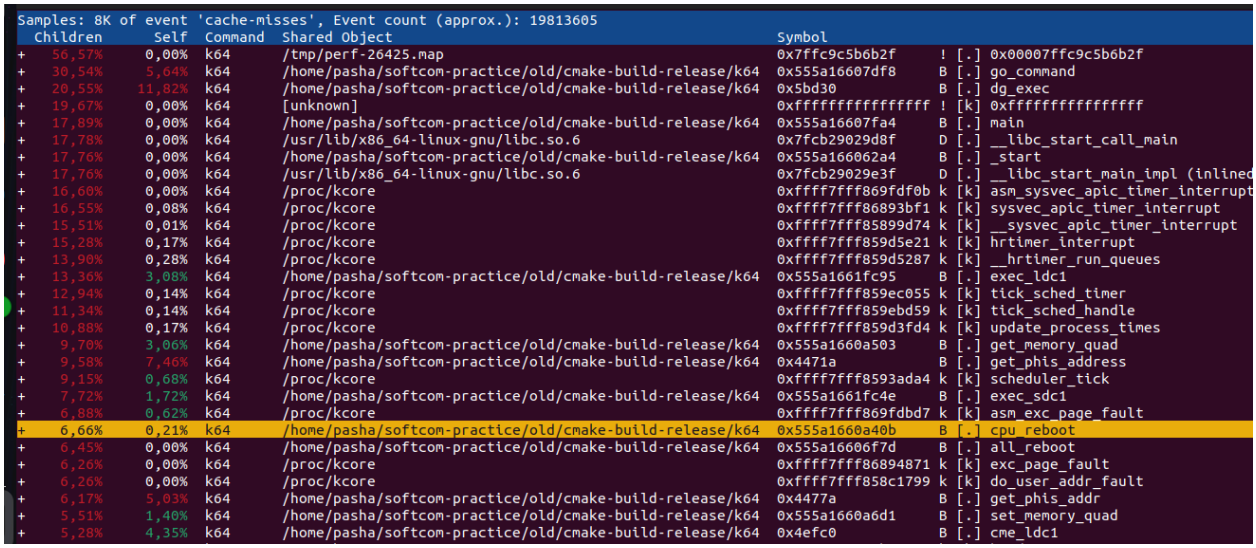
Интересным наблюдением является то, что данные, полученные на процессоре Intel, сильно разнятся с данными, полученными на AMD. В случае Intel было выявлено, что сильнее всего от промахов кэша страдает функция cpu_reboot. Подробная таблица представлена на рисунке 3.

```
Samples: 8K of event 'cache-misses', Event count (approx.): 16225399
  Children      Self  Command  Shared Object                                  Symbol
+   81,25%     0,00%  k64      /home/lich/work/softcom/old.old/build2/k64     0x56172dcc02e4     d [.] _start
+   81,25%     0,00%  k64      /usr/lib/x86_64-linux-gnu/libc.so.6            0x7f07d270de3f     b [.] __libc_start_main_impl (inlined)
+   81,25%     0,00%  k64      /usr/lib/x86_64-linux-gnu/libc.so.6            0x7f07d270dd8f     b [.] __libc_start_call_main
+   81,25%     0,00%  k64      /home/lich/work/softcom/old.old/build2/k64     0x56172dcc371d     d [.] main
+   78,65%    54,53%  k64      /home/lich/work/softcom/old.old/build2/k64     0x56172dcc5e4b     d [.] cpu_reboot
+   54,53%     0,00%  k64      /home/lich/work/softcom/old.old/build2/k64     0x56172dcc2d74     d [.] load_elf_file
+   28,20%     0,00%  k64      /home/lich/work/softcom/old.old/build2/k64     0x56172dcdd0d7     d [.] load_elf
+   24,12%     0,00%  k64      /home/lich/work/softcom/old.old/build2/k64     0x56172dcc2514     d [.] all_reboot
+   24,12%     0,00%  k64      [kernel.kallsyms]                              0xffffffffb6200bb7 ! [k] 0xffffffffb6200bb7
+   24,12%     0,00%  k64      [kernel.kallsyms]                              0xffffffffb61bb647 ! [k] 0xffffffffb61bb647
+   24,12%     0,00%  k64      [kernel.kallsyms]                              0xffffffffb549f839 ! [k] 0xffffffffb549f839
+   24,12%     0,00%  k64      [kernel.kallsyms]                              0xffffffffb5705348 ! [k] 0xffffffffb5705348
+   24,12%     0,00%  k64      [kernel.kallsyms]                              0xffffffffb5704f85 ! [k] 0xffffffffb5704f85
+   24,12%     0,00%  k64      [kernel.kallsyms]                              0xffffffffb5702fea ! [k] 0xffffffffb5702fea
+   14,92%     0,00%  k64      [kernel.kallsyms]                              0xffffffffb56feba4 ! [k] 0xffffffffb56feba4
+   14,49%     0,00%  k64      [kernel.kallsyms]                              0xffffffffb577e16d ! [k] 0xffffffffb577e16d
+   11,78%     0,00%  k64      /tmp/perf-115441.map                           0x7ffff61c491f     ! [.] 0x00007ffff61c491f
+    7,73%     2,00%  k64      /home/lich/work/softcom/old.old/build2/k64     0x56172dcc3545     d [.] go_command
+    7,33%     0,00%  k64      [kernel.kallsyms]                              0xffffffffb577be15 ! [k] 0xffffffffb577be15
+    5,15%     0,00%  k64      [unknown]                                      0xffffffffffffffff ! [.] 0xffffffffffffffff
```

Рис. 3: Промахи кэша на процессоре Intel.

В случае AMD промахов кэша в той же функции cpu_reboot на порядок меньше. При этом сами промахи распределены равномерно между

большим количеством функций, тогда как в первом случае более половины всех промахов приходится на cpu_reboot. Объяснить это можно тем, что поведение соответствующих модулей мониторинга производительностью в процессорах отличается. Подробная таблица промахов кэша на процессоре AMD представлена на рисунке 4.



Samples: 8K of event 'cache-misses', Event count (approx.): 19813605

	Children	Self	Command	Shared Object	Symbol
+	56,57%	0,00%	k64	/tmp/perf-26425.map	0x7ffc9c5b6b2f ! [.] 0x00007ffc9c5b6b2f
+	30,54%	5,64%	k64	/home/pasha/softcom-practice/old/cmake-build-release/k64	0x555a16607df8 B [.] go_command
+	20,55%	11,82%	k64	/home/pasha/softcom-practice/old/cmake-build-release/k64	0x5bd30 B [.] dg_exec
+	19,67%	0,00%	k64	[unknown]	0xffffffffffffffff ! [k] 0xffffffffffffffff
+	17,89%	0,00%	k64	/home/pasha/softcom-practice/old/cmake-build-release/k64	0x555a16607fa4 B [.] main
+	17,78%	0,00%	k64	/usr/lib/x86_64-linux-gnu/libc.so.6	0x7fcb29029d8f D [.] __libc_start_call_main
+	17,76%	0,00%	k64	/home/pasha/softcom-practice/old/cmake-build-release/k64	0x555a166062a4 B [.] _start
+	17,76%	0,00%	k64	/usr/lib/x86_64-linux-gnu/libc.so.6	0x7fcb29029e3f D [.] __libc_start_main_impl (inlined
+	16,60%	0,00%	k64	/proc/kcore	0xffff7fff869fdf0b k [k] asm_sysvec_apic_timer_interrupt
+	16,55%	0,08%	k64	/proc/kcore	0xffff7fff86893bf1 k [k] sysvec_apic_timer_interrupt
+	15,51%	0,01%	k64	/proc/kcore	0xffff7fff85899d74 k [k] __sysvec_apic_timer_interrupt
+	15,28%	0,17%	k64	/proc/kcore	0xffff7fff859d5e21 k [k] hrtimer_interrupt
+	13,90%	0,28%	k64	/proc/kcore	0xffff7fff859d5287 k [k] __hrtimer_run_queues
+	13,36%	3,08%	k64	/home/pasha/softcom-practice/old/cmake-build-release/k64	0x555a1661fc95 B [.] exec_ldc1
+	12,94%	0,14%	k64	/proc/kcore	0xffff7fff859ec055 k [k] tick_sched_timer
+	11,34%	0,14%	k64	/proc/kcore	0xffff7fff859ebd59 k [k] tick_sched_handle
+	10,88%	0,17%	k64	/proc/kcore	0xffff7fff859d3fd4 k [k] update_process_times
+	9,70%	3,06%	k64	/home/pasha/softcom-practice/old/cmake-build-release/k64	0x555a1660a503 B [.] get_memory_quad
+	9,58%	7,46%	k64	/home/pasha/softcom-practice/old/cmake-build-release/k64	0x4471a B [.] get_phis_address
+	9,15%	0,68%	k64	/proc/kcore	0xffff7fff8593ada4 k [k] scheduler_tick
+	7,72%	1,72%	k64	/home/pasha/softcom-practice/old/cmake-build-release/k64	0x555a1661fc4e B [.] exec_sdc1
+	6,88%	0,62%	k64	/proc/kcore	0xffff7fff869fdbd7 k [k] asm_exc_page_fault
+	6,66%	0,21%	k64	/home/pasha/softcom-practice/old/cmake-build-release/k64	0x555a1660a40b B [.] cpu_reboot
+	6,45%	0,00%	k64	/home/pasha/softcom-practice/old/cmake-build-release/k64	0x555a16606f7d B [.] all_reboot
+	6,26%	0,00%	k64	/proc/kcore	0xffff7fff86894871 k [k] exc_page_fault
+	6,26%	0,00%	k64	/proc/kcore	0xffff7fff858c1799 k [k] do_user_addr_fault
+	6,17%	5,03%	k64	/home/pasha/softcom-practice/old/cmake-build-release/k64	0x4477a B [.] get_phis_addr
+	5,51%	1,40%	k64	/home/pasha/softcom-practice/old/cmake-build-release/k64	0x555a1660a6d1 B [.] set_memory_quad
+	5,28%	4,35%	k64	/home/pasha/softcom-practice/old/cmake-build-release/k64	0x4efc0 B [.] cme_ldc1

Рис. 4: Промахи кэша на процессоре AMD.

Из-за большого количества промахов кэша в функции cpu_reboot на процессоре Intel было принято решение проанализировать её реализацию. Её фрагмент представлен в листинге 6.

Листинг 6: Фрагмент функции cpu_reboot.

```
cpu_reboot:
  ...
  movabs $ram_memory, RDI
  mov $(MEMORY_SIZE*1024*1024/8), ECX
  xor RAX, RAX
  rep stosq
  movabs $mmu_table1, RDI
  mov $(1024*1024), ECX
  mov $-1, EAX
  rep stosd
  movabs $mmu_table2, RDI
  mov $(MEMORY_SIZE*256), ECX
  mov $-1, EAX
  rep stosd
```

```
  ...
```

Благодаря технологии enhanced rep movsb (ERMSB)[4], на процессорах Intel и AMD в случае манипуляции большими объёмами памяти команда rep stosb работает гораздо быстрее, чем rep stosq или rep stosd. Поэтому было принято решение заменить приведённые выше команды на rep stosb:

Листинг 7: Фрагмент функции cpu_reboot с внесёнными изменениями.

```
cpu_reboot:
  ...
    movabs $ram_memory, RDI
    mov $(MEMORY_SIZE*1024*1024), ECX
    xor RAX, RAX
    rep stosb
    movabs $mmu_table1, RDI
    mov $(1024*1024*4), ECX
    mov $-1, EAX
    rep stosb
    movabs $mmu_table2, RDI
    mov $(MEMORY_SIZE*256*4), ECX
    mov $-1, EAX
    rep stosb
  ...
```

[4]Технология, заключающаяся в том, что при работе со сторовыми операциями rep способен достигать высоких показателей производительности с помощью mosvb и stosb.

4. Эксперимент

Тестовый стенд эксперимента:

- CPU: AMD(R) Ryzen 7 5800H with Radeon Graphics @ 3.20GHz;
- 16 ГБ ОЗУ.

Были проведены тестовые замеры следующих сценариев:

- исполнение функции cpu_reboot двести раз в цикле до применения технологии ERMSB;
- исполнение функции cpu_reboot двести раз в цикле после применения технологии ERMSB.

Оба тестовых сценария были обработаны по двадцать раз. В результате экспериментов по полученным данным были построены 95% доверительные интервалы для среднего. Итоговые интервалы:

- 1,551±0,018 секунд до улучшений;
- 1,511±0,014 секунд после улучшений.

Замеры показывают, что разница статистически значима, поэтому ускорение имеет место.

Заключение

Приложение является проектом с закрытым исходным кодом.

В ходе выполнения данной работы были достигнуты все поставленные цели, а именно:

1. приложение было перенесено под Linux, возможность его сборки под Windows сохранена;
2. DWARF метки были расставлены;
3. построен FlameGraph для анализа приложения k64;
4. получилось ускорить приложение k64 с использованием ERMSB.

Список литературы

[1] AMD uProf. — URL: https://www.amd.com/en/developer/uprof.html (дата обращения: 4 января 2024 г.).

[2] Aarno Daniel, Engblom Jakob. Software and system development using virtual platforms : full-system simulation with Wind River Simics. — Morgan Kaufmann, Elsevier Inc, 2015. — ISBN: 9780128008133.

[3] CFI directives. — URL: https://sourceware.org/binutils/docs/as/CFI-directives.html (дата обращения: 3 января 2024 г.).

[4] CFI directives in assembly files manual. — URL: https://www.imperialviolet.org/2017/01/18/cfi.html (дата обращения: 4 января 2024 г.).

[5] Flame Graph. — URL: https://github.com/brendangregg/FlameGraph (дата обращения: 3 января 2024 г.).

[6] Gprof: the GNU profiler. — URL: https://sourceware.org/binutils/docs/gprof/ (дата обращения: 3 января 2024 г.).

[7] Intel VTune Profiler. — URL: https://hpc-wiki.info/hpc/Intel_VTune (дата обращения: 4 января 2024 г.).

[8] KOMDIV-64 processor. — URL: https://ru.wikipedia.org/wiki/KOMDIV-64 (дата обращения: 3 января 2024 г.).

[9] Linpack benchmark. — URL: https://en.wikipedia.org/wiki/LINPACK_benchmarks (дата обращения: 5 января 2024 г.).

[10] List of performance analysis tools. — URL: https://en.wikipedia.org/wiki/List_of_performance_analysis_tools (дата обращения: 4 января 2024 г.).

[11] Perf: Linux manual page. — URL: https://man7.org/linux/man-pages/man1/perf.1.html (дата обращения: 3 января 2024 г.).

[12] System V Application Binary Interface for x86_64. — URL: https://wiki.osdev.org/System_V_ABI (дата обращения: 4 января 2024 г.).

[13] Valgrind. — URL: https://valgrind.org (дата обращения: 3 января 2024 г.).