

Параллельное программирование для высокопроизводительных систем

сентябрь – декабрь 2021 г.

Лектор доцент Попова Нина Николаевна

Лекция 4

7 октября 2021 г.

Тема

- Классы архитектур современных процессоров.
- Использование аппаратных счетчиков для анализа производительности программ

Архитектура процессоров

- 3 основных класса:
 - **CISC** (Complex Instruction Set)
 - **RISC** (Restricted (Reduced) Instruction Set Computer)
 - **MISC** (Multipurpose Instruction Set Computer)

CISC

CISC (Complex Instruction Set Computer — «компьютер с полным набором команд»)

- арифметические действия выполняются одной командой;
- нефиксированная длина команд;
- каждый регистр выполняет строго свою функцию и их количество ограничено
- Самый яркий пример CISC архитектуры — x86 (он же IA-32) и x86_64 (он же AMD64).
- Недостатки
 - **сложны в проектировании** и дороги в производстве
 - **проблемы с полноценным распараллеливанием** вычислений (приходится постоянно **оптимизировать софт**).

RISC

RISC (Reduced Instruction Set Computer — «компьютер с сокращённым набором команд»)

Архитектура процессора, в котором быстродействие увеличивается за счёт упрощения инструкций: их декодирование становится более простым, а время выполнения — меньшим. Первые RISC-процессоры не имели даже инструкций умножения и деления и не поддерживали работу с числами с плавающей запятой.

RISC

RISC

- Конвейерные функциональные устройства (ФУ)
- Несколько ФУ на процессоре
- Спекулятивное исполнение операций
- Многоуровневый кэш
- Блоки (Cache lines) могут разделяться между несколькими процессорами

RISC

- Начало исследований данной области положено компанией **IBM** (в исследовательском центре **IBM**, имени **Томаса Джона Уотсона**) в **1975** году.
- 40% рынка в настоящее время
- RISC-инструкции просты, для их выполнения нужно меньше логических элементов, что в конечном итоге снижает стоимость процессора. Но большая часть программного обеспечения сегодня написана и откомпилирована специально для CISC-процессоров фирмы Intel. Для использования архитектуры RISC нынешние программы должны быть перекомпилированы, а иногда и переписаны заново.

RISC

- По сравнению с CISC эта архитектура имеет константную длину команды, а также меньшее количество схожих инструкций, позволяя уменьшить итоговую цену процессора и энергопотребление, что критично для мобильного сегмента. У RISC также большее количество регистров.
- Примеры RISC-архитектур: PowerPC, серия архитектур ARM (ARM7, ARM9, ARM11, Cortex).
- В общем случае RISC быстрее CISC. Даже если системе RISC приходится выполнять 4 или 5 команд вместо одной, которую выполняет CISC, RISC все равно выигрывает в скорости, так как RISC-команды выполняются в 10 раз быстрее.

MISC

MISC (Minimal Instruction Set Computer — «компьютер с минимальным набором команд»)

Ещё более простая архитектура, используемая в первую очередь для ещё большего уменьшения итоговой цены и энергопотребления процессора. Используется в IoT-сегменте и недорогих компьютерах, например, роутерах.

Характеризуются сложностью написания программ.

VLIW

VLIW (Very Long Instruction Word — «очень длинная машинная команда»)

Архитектура процессоров с несколькими вычислительными устройствами. Характеризуется тем, что одна инструкция процессора содержит несколько операций, которые должны выполняться параллельно.

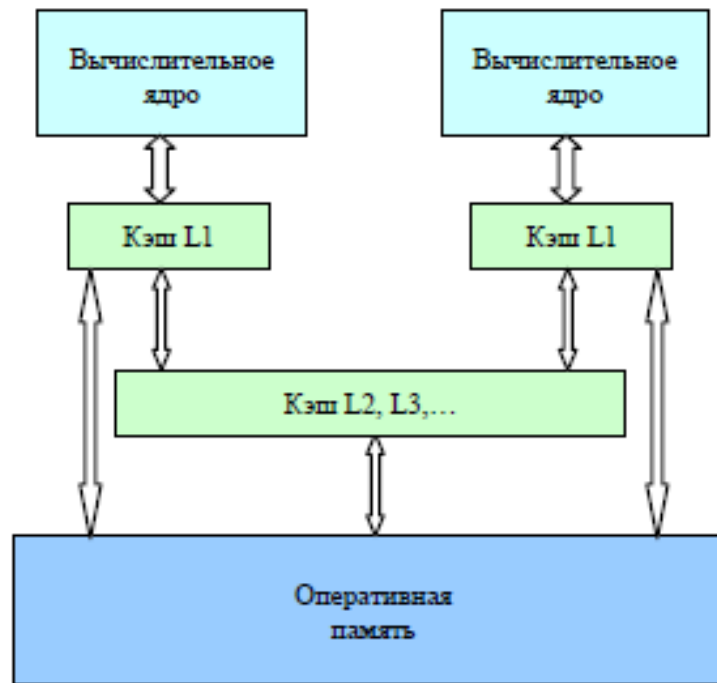
По сути является архитектурой CISC со своим аналогом спекулятивного исполнения команд, только сама спекуляция выполняется во время компиляции, а не во время работы программы.

Примеры архитектуры: Intel Itanium, Эльбрус-3.

Многоядерные процессоры

- Возврат к более «простым» процессорам с более низкой тактовой частотой и менее сложной логикой реализации. Процессоры становятся менее энергоемкими, более простыми для изготовления и, как результат, более надежными.
- Реализация в единственном кремниевом кристалле нескольких **вычислительных ядер** в составе одного многоядерного процессора, при этом по своим вычислительным возможностям эти ядра могут не уступать обычным (одноядерным) процессорам.

Пример организации 2-ух ядерного процессора



Одновременная многопоточность (*simultaneous multithreading, SMT*)

- Предложена в 1995 г. Дином Тулсенем (Dean Tullsen) и позднее активно развита компанией Интел под названием технологии *гиперпоточности* (*hyper threading, HT*).
- В рамках такого подхода процессор дополняется средствами запоминания состояния потоков, схемами контроля одновременного выполнения нескольких потоков и т. д. За счет этих дополнительных средств на активной стадии выполнения может находиться несколько потоков; при этом одновременно выполняемые потоки конкурируют за исполнительные блоки единственного процессора и, как результат, выполнение отдельных потоков может блокироваться, если требуемые в данный момент времени блоки процессора оказываются уже задействованными.

Одновременная многопоточность (*simultaneous multithreading, SMT*)

- Как правило, число аппаратно-поддерживаемых потоков равно 2, 4 и даже 8.
- Аппаратно-поддерживаемые потоки на логическом уровне операционных систем воспринимаются как отдельные процессоры
- Использование процессоров с поддержкой многопоточности может приводить к существенному ускорению вычислений. Так, имеется большое количество примеров, показывающих, что на процессорах компании Интел с поддержкой технологии гиперпоточности достигается повышение скорости вычислений около 30%.

Тема

Использование аппаратных счетчиков для анализа производительности программ

Мотивация

При разработке эффективных алгоритмов необходимо учитывать:

- поведение кеш-памяти
- ограничения по используемой оперативной памяти и других ресурсов
- эффективность вещественных операций
- поведение ветвлений

Метрики производительности процессоров

- Теоретическая пиковая производительность R_{theor} : maximum FLOPS, которые могут быть достигнуты теоретически.
 - $\text{Clock_rate} \times \#\text{cpus} \times \#\text{FPU/CPU}$
 - 3GHz, 2 cpus, 1 FPU/CPU $\rightarrow R_{\text{theor}} = 3 \times 10^9 \times 2 = 6$ GFLOPS
- Реальная производительность R_{real} : FLOPS на определенных операциях, например, векторном умножении
- Sustained performance $R_{\text{sustained}}$: производительность, полученная для конкретных приложений

$$R_{\text{sustained}} \ll R_{\text{real}} \ll R_{\text{theor}}$$

Типично: $R_{\text{sustained}} < 10\% R_{\text{theor}}$

Аппаратные счетчики

- Разработчики аппаратуры добавили в устройство процессоров специальные регистры для измерения различных характеристик микропроцессора
- В общем случае такие аппаратные счетчики позволяют измерять:
 - время
 - кеш и ветвление
 - шаблоны доступа к памяти
 - поведение конвейерного выполнения операций
 - производительность вещественных операций
 - счетчики выполненных инструкций
 -

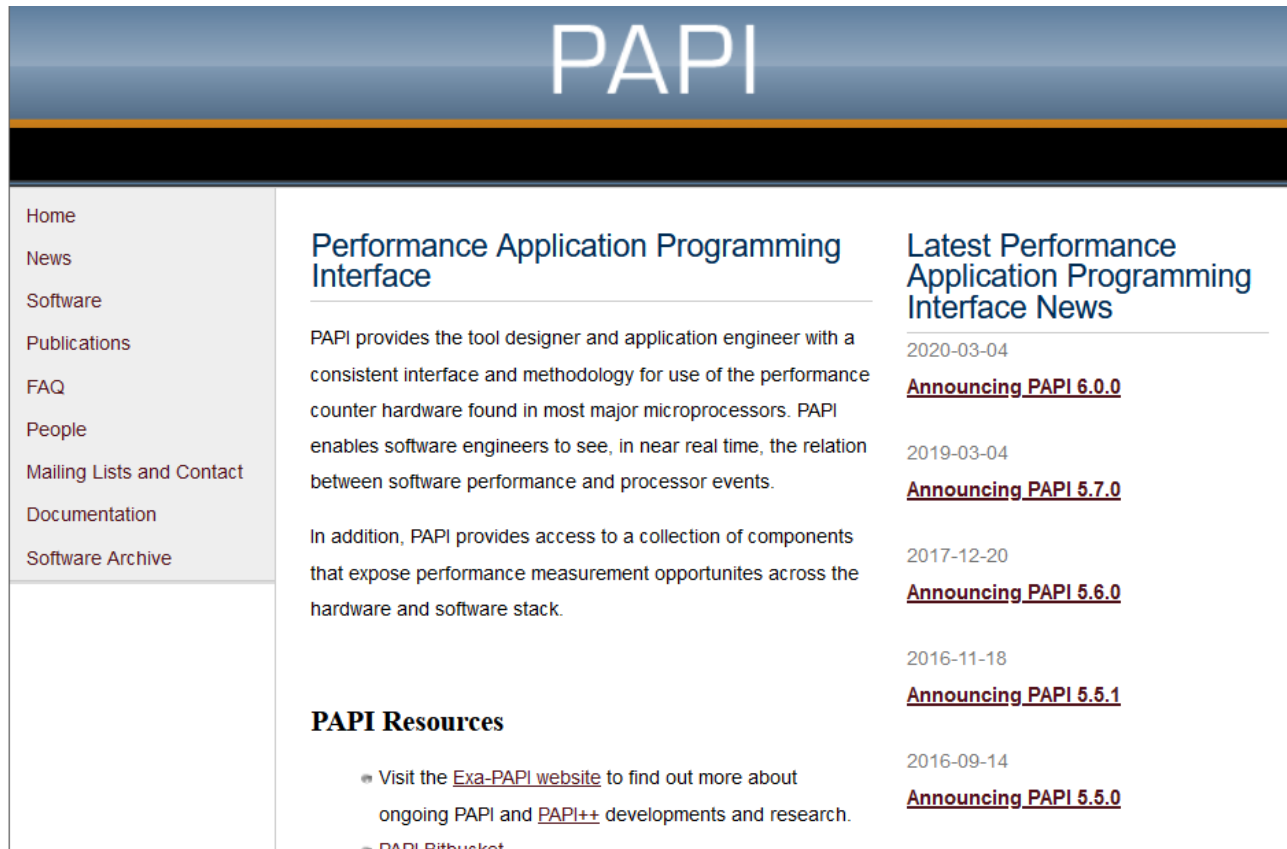
Аппаратные метрики

- Отношение числа тактов к числу инструкций IPC
- Число инструкций с плавающей точкой FLOPS
- Интенсивность целочисленных инструкций
- Отношение числа промахов кэша для операций чтения/записи к общему числу промахов кэша
- Число промахов кэша
- Число промахов кэша для операций чтения
- TLB промахи

PAPI (<http://icl.cs.utk.edu/papi/>)

- **Performance Application Programming Interface**
- Цель создания PAPI – разработка стандартизованного, переносимого и эффективного доступа к аппаратным счетчикам современных процессоров
- Parallel Tools Consortium проект
<http://www.ptools.org/>
- Текущая версия PAPI 6.0.0 (март 2020)
- Exa-PAPI

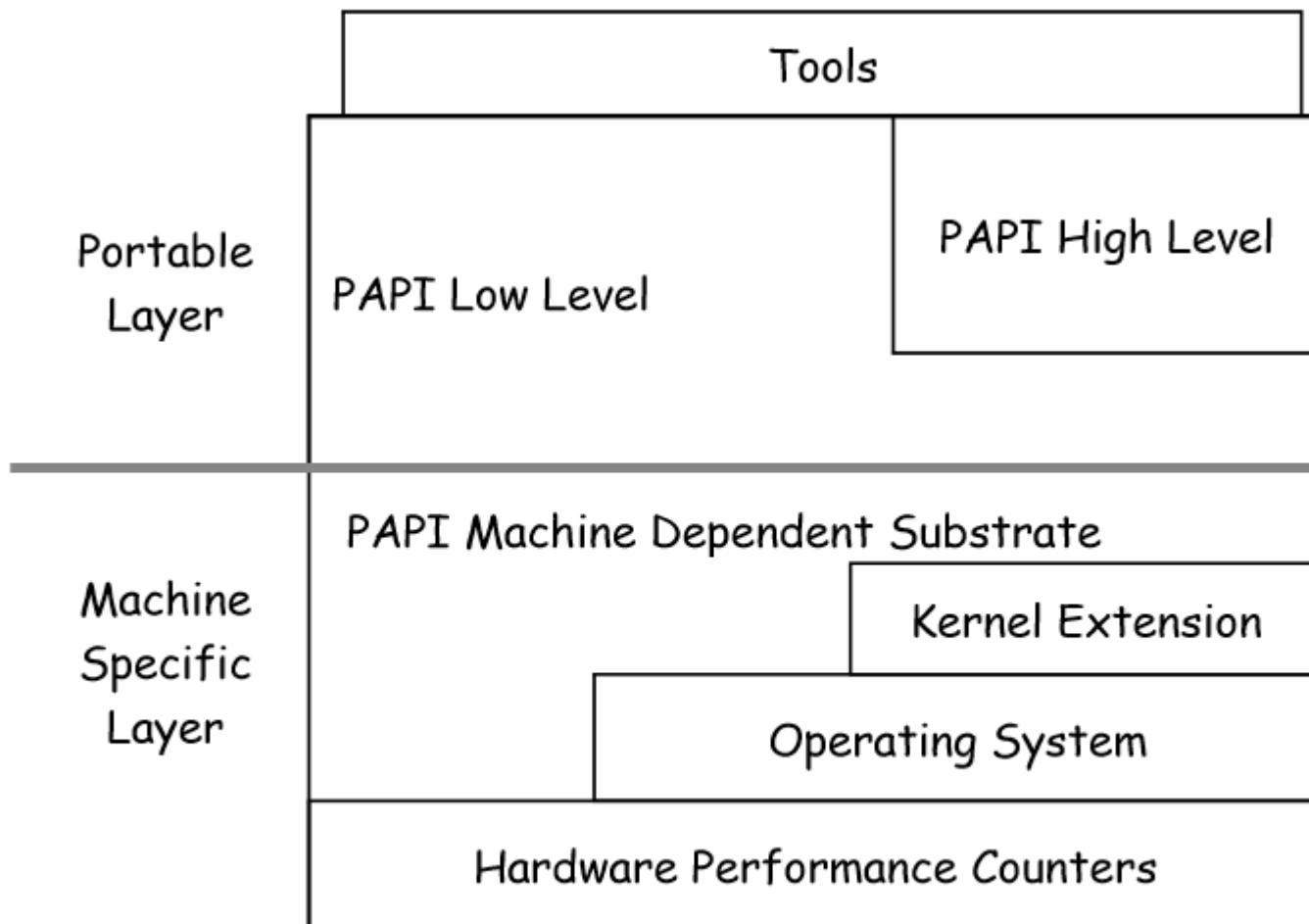
Сайт PAPI



РАPI

- Интерфейс для работы со счетчиками производительности
 - ✓ Минимальные накладные расходы
 - ✓ Переносимость на различные платформы
 - ✓ Программные , C и Fortran API
 - ✓ Информация о системе и счетчиках

Структура PAPI



Интерфейсы доступа к счетчикам RAPI

- RAPI обеспечивает 3 типа интерфейса к аппаратным счетчикам:
 1. Интерфейс нижнего уровня для аппаратных событий, объединенных пользователем в группы (*EventSets*).
 2. Высокоуровневый интерфейс, обеспечивающий возможности стартовать, прекращать сбор событий и считывать счетчики для заданного списка событий
 3. Графический интерфейс для визуализации собранной информации.

События RAPI

- **Событие** – появление специфических сигналов, связанных с функционированием аппаратуры.

- **Hardware performance counters** – небольшое множество регистров, фиксирующих число возникающих событий, например, промахи кеша, вещественные операции и др.

Каждый процессор имеет свой конкретный способ реализации событий. RAPI обеспечивает универсальный доступ к таким событиям


События PAPI

- **Preset** (предустановленные) события
 - События, предназначенные для любых платформ
 - PAPI_TOT_INS
- **Native** события
 - Платформенно зависимые события
 - L3_CACHE_MISS
- **Derived** (производные) события
 - Preset события, определенные через несколько native событий
 - PAPI_L1_TCM может быть L1 data misses + L1 instruction misses

Preset события

- Приблизительно 100 preset событий.
- Определены в заголовочном файле `papiStdEventDefs.h`.

Утилиты PAPI

 popova@polus-ib:~/SSPP_2021/Lect_PAPI

```
[popova@polus-ib Lect_PAPI]$ ls /usr/bin/papi*  
/usr/bin/papi_avail          /usr/bin/papi_event_chooser  
/usr/bin/papi_clockres      /usr/bin/papi_mem_info  
/usr/bin/papi_command_line  /usr/bin/papi_multiplex_cost  
/usr/bin/papi_component_avail /usr/bin/papi_native_avail  
/usr/bin/papi_cost          /usr/bin/papi_version  
/usr/bin/papi_decode        /usr/bin/papi_xml_event_info  
/usr/bin/papi_error_codes  
[popova@polus-ib Lect_PAPI]$
```

Утилита papi_avail (1)

```
[popova@polus-ib Lect_PAPI]$ /usr/bin/papi_avail
Available events and hardware information.
-----
PAPI Version           : 5.2.0.0
Vendor string and code : (0)
Model string and code  : POWER8NVL (8335)
CPU Revision           : 1.000000
CPU Max Megahertz      : 4023
CPU Min Megahertz      : 2061
Hdw Threads per core   : 8
Cores per Socket      : 10
Sockets                : 2
NUMA Nodes             : 2
CPUs per Node          : 80
Total CPUs             : 160
Running in a VM        : no
Number Hardware Counters : 6
Max Multiplex Counters : 64
-----

  Name      Code      Avail Deriv Description (Note)
PAPI_L1_DCM 0x80000000 Yes   Yes  Level 1 data cache misses
PAPI_L1_ICM 0x80000001 Yes   No   Level 1 instruction cache misses
```

Утилита papi_avail (2)

Name	Code	Avail	Deriv	Description (Note)
PAPI_L1_DCM	0x80000000	Yes	Yes	Level 1 data cache misses
PAPI_L1_ICM	0x80000001	Yes	No	Level 1 instruction cache misses
PAPI_L2_DCM	0x80000002	Yes	No	Level 2 data cache misses
PAPI_L2_ICM	0x80000003	No	No	Level 2 instruction cache misses
PAPI_L3_DCM	0x80000004	No	No	Level 3 data cache misses
PAPI_L3_ICM	0x80000005	Yes	No	Level 3 instruction cache misses
PAPI_L1_TCM	0x80000006	No	No	Level 1 cache misses
PAPI_L2_TCM	0x80000007	No	No	Level 2 cache misses
PAPI_L3_TCM	0x80000008	No	No	Level 3 cache misses
PAPI_CA_SNP	0x80000009	No	No	Requests for a snoop
PAPI_CA_SHR	0x8000000a	No	No	Requests for exclusive access to shared cache line
PAPI_CA_CLN	0x8000000b	No	No	Requests for exclusive access to clean cache line
PAPI_CA_INV	0x8000000c	No	No	Requests for cache line invalidation
PAPI_CA_ITV	0x8000000d	No	No	Requests for cache line intervention
PAPI_L3_LDM	0x8000000e	No	No	Level 3 load misses
PAPI_L3_STM	0x8000000f	No	No	Level 3 store misses
PAPI_BRU_IDL	0x80000010	No	No	Cycles branch units are idle
PAPI_FXU_IDL	0x80000011	No	No	Cycles integer units are idle

Запрос событий

Функции интерфейса нижнего уровня для запроса существования preset или native события (поддерживает ли аппаратура заданное событие) и выяснения деталей о событии:

PAPI_query_event(EventCode)

PAPI_get_event_info(EventCode, &info)

PAPI_enum_event(&EventCode, modifier)

Основы PAPI

PAPI_start_counters

PAPI_stop_counters

Основы PAPI. Пример.

```
#include "papi.h"
#define NUM_EVENTS 2
long long values[NUM_EVENTS];
unsigned int Events [NUM_EVENTS]=
{PAPI_TOT_INS,PAPI_TOT_CYC};

/* Start the counters */
PAPI_start_counters((int*)Events,NUM_EVENTS);
/* What we are monitoring... */
do_work();
/* Stop counters and store results in values */
retval = PAPI_stop_counters(values,NUM_EVENTS);
```

Основные функции PAPI

- Восемь основных функций:

- *PAPI_num_counters*
- *PAPI_start_counters*,
- *PAPI_stop_counters*
- *PAPI_read_counters*
- *PAPI_accum_counters*
- *PAPI_flops*
- *PAPI_flips*, *PAPI_ipc*

Пример 2 (1)

```
#include <papi.h>
#define NUM_FLOPS 10000
#define NUM_EVENTS 1
main() {
    int Events[NUM_EVENTS] = {PAPI_TOT_INS};
    long_long values[NUM_EVENTS];
    /* Start counting events */
    if (PAPI_start_counters(Events, NUM_EVENTS) != PAPI_OK)
        handle_error(1);
```

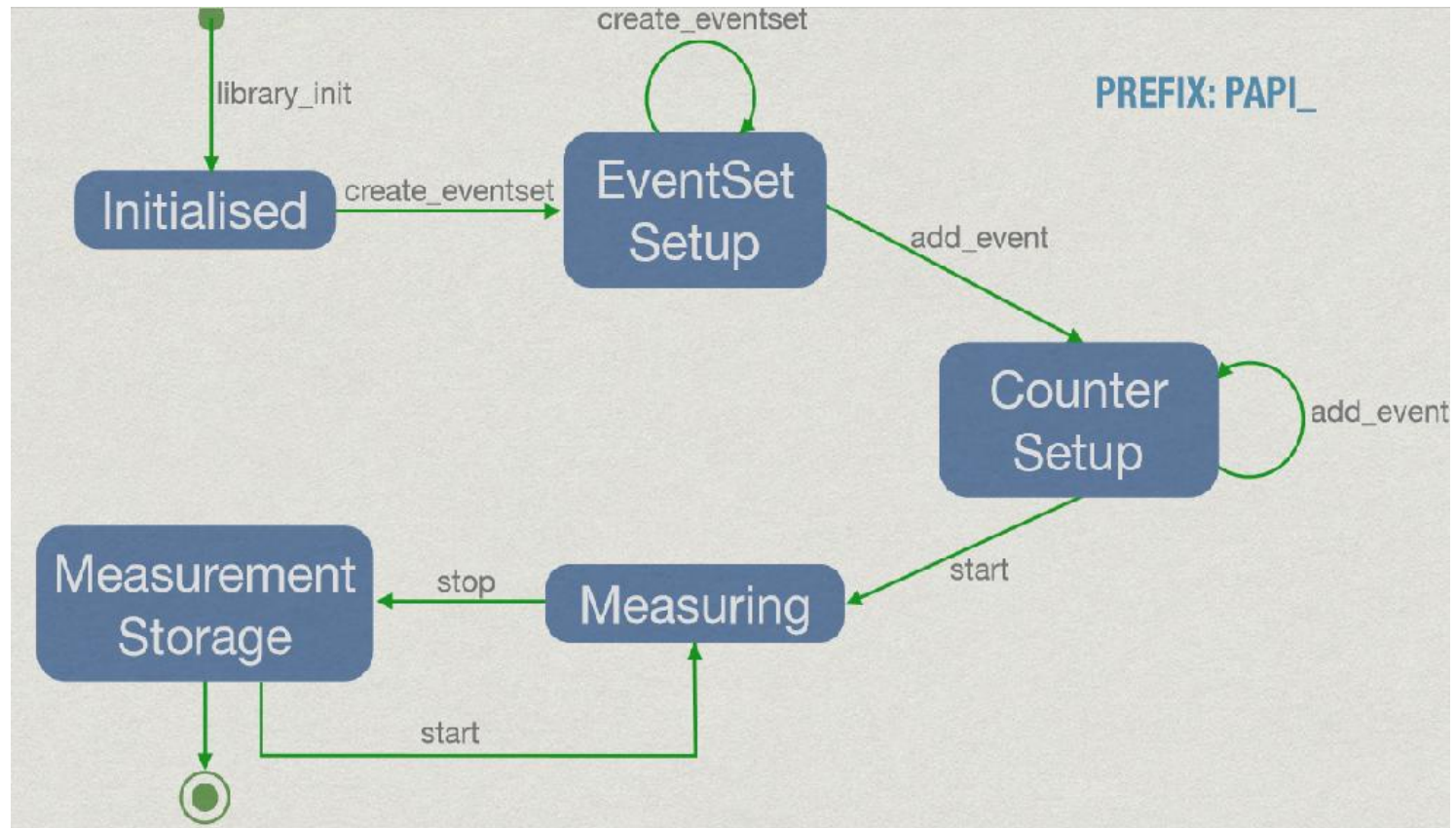
Пример 2 (2)

```
/* Defined in tests/do_loops.c in the PAPI source distribution */
do_flops(NUM_FLOPS);
/* Read the counters */
if (PAPI_read_counters (values, NUM_EVENTS) != PAPI_OK)
    handle_error(1);
printf("After reading the counters: %lld\n", values[0]);
do_flops(NUM_FLOPS);
/* Add the counters */
if (PAPI_accum_counters(values, NUM_EVENTS) != PAPI_OK)
    handle_error(1);
printf("After adding the counters: %lld\n", values[0]);
```

Пример 2 (3)

```
do_flops(NUM_FLOPS);  
/* Stop counting events */  
if (PAPI_stop_counters(values, NUM_EVENTS) != PAPI_OK)  
    handle_error(1);  
    printf("After stopping the counters: %lld\n", values[0]);  
  
}
```

Общая схема



Низкоуровневое API

- Повышает эффективность и функциональность по сравнению с high level PAPI interface
 - 54 функции
 - обеспечивает доступ к native событиям
 - поддерживает информацию об исполняемом коде, используемой аппаратуре и памяти
 - имеет опции для мультиплексирования событий и обработки переполнения счетчиков

Множества событий (Event set)

- Множество событий (event set) содержит ключевую информацию о
 - Используемых низкоуровневых аппаратных счетчиках
 - Актуальном значении считанных аппаратных счетчиков
 - Состоянии множества событий (running/not running)
 - Используемых опциях (например, granularity, overflow, profiling)
- Могут пересекаться, если используют одинаковые установки аппаратных счетчиков.
 - Позволяют проводить inclusive/exclusive измерения

Функции для установки множества событий

- Event set management
PAPI_create_eventset, PAPI_add_event[s],
PAPI_rem_event[s], PAPI_destroy_eventset
- Event set control
PAPI_start, PAPI_stop, PAPI_read, PAPI_accum
- Event set inquiry
PAPI_query_event, PAPI_list_events,...

Пример 3

```
#include "papi.h"
#define NUM_EVENTS 2
int
    Events[NUM_EVENTS]={PAPI_FP_INS,PAPI_TOT_CYC},EventSet=PAPI_NULL;
long_long values[NUM_EVENTS];
/* Initialize the Library */
retval = PAPI_library_init(PAPI_VER_CURRENT);
/* Allocate space for the new eventset and do setup */
retval = PAPI_create_eventset(&EventSet);
/* Add Flops and total cycles to the eventset */
retval = PAPI_add_events(&EventSet,Events,NUM_EVENTS);
/* Start the counters */
retval = PAPI_start(EventSet);

do_work(); /* What we want to monitor*/

/*Stop counters and store results in values */
retval = PAPI_stop(EventSet,values);
```

Инициализация библиотеки PAPI

PAPI_library_init()

```
PAPI_library_init();
```

```
if (PAPI_VER_CURRENT !=  
    PAPI_library_init(PAPI_VER_CURRENT))  
    ehandler("PAPI_library_init error.");
```

PAPI_num_counters()

Проверка числа счетчиков, которые могут контролироваться данным CPU

```
const size_t EVENT_MAX = PAPI_num_counters();
```

PAPI_query_event()

```
if (PAPI_OK!=PAPI_query_event(PAPI_TOT_INS))
    ehandler("Cannot count PAPI_TOT_INS.");
if (PAPI_OK != PAPI_query_event(PAPI_L1_DCM))
    ehandler("Cannot count PAPI_L1_DCM.");
if (PAPI_OK != PAPI_query_event(PAPI_L2_DCM))
    ehandler("Cannot count PAPI_L2_DCM.");
```

PAPI_start_counters()

```
size_t EVENT_COUNT = 3;  
int events[] = { PAPI_TOT_INS, PAPI_L1_DCM, PAPI_L2_DCM };  
PAPI_start_counters(events, EVENT_COUNT);
```

PAPI_read_counters()

```
long long values[EVENT_COUNT];  
if (PAPI_OK != PAPI_read_counters(values, EVENT_COUNT))  
    ehandler("Problem reading counters 1.");  
C = matrix_prod(n, n, n, n, A, B);  
if (PAPI_OK != PAPI_read_counters(values, EVENT_COUNT))  
    ehandler("Problem reading counters 2.");  
printf("%d %lld %lld %lld\n", n, values[0], values[1], values[2])
```

Функция PAPI_flops

```
int PAPI_flops (float *rtime, float *ptime, long_long *flpops,  
               float *mflops);
```

Упрощает получение Mflops/s, real and processor time

Параметры:

rtime -- total realtime since the first PAPI_flops() call

ptime -- total process time since the first PAPI_flops() call

flpops -- total floating point operations since the first call

mflops -- Mflop/s achieved since the previous call

Пример использования PAPI_flops()

```
float rtime;  
float ptime;  
long long flpops;  
float mflops;  
if (PAPI_OK != PAPI_flops(&rtime, &ptime, &flpops, &mflops))  
    ehandler("Problem reading flops 1");  
C = matrix_prod(n, n, n, n, A, B);  
if (PAPI_OK != PAPI_flops(&rtime, &ptime, &flpops, &mflops))  
    ehandler("Problem reading flops 2");  
printf("%d %lld %f\n", n, flpops, mflops);
```

Простой пример

```
#include "papi.h"
#define NUM_EVENTS 2
int Events[NUM_EVENTS]={PAPI_FP_INS,PAPI_TOT_CYC}, EventSet;
long_long values[NUM_EVENTS];
/* Initialize the Library */
retval = PAPI_library_init(PAPI_VER_CURRENT);
/* Allocate space for the new eventset and do setup */
retval = PAPI_create_eventset(&EventSet);
/* Add Flops and total cycles to the eventset */
retval = PAPI_add_events(&EventSet,Events,NUM_EVENTS);
/* Start the counters */
retval = PAPI_start(EventSet);

do_work(); /* What we want to monitor*/

/*Stop counters and store results in values */
retval = PAPI_stop(EventSet,values);
```

Пример анализа

