Second Lab Assignment: System Modeling and Profiling

STUDENTS IDENTIFICATION:

Number:	Name:
106059	Lara Alexandra Gomes de Faria
106422	Leonor Nunes da Miranda Gonçalves Francisco
106171	Guilherme Vaz Rocha

2 Exercise

Please justify all your answers with values from the experiments. Stride = 4096

1. What is the cache capacity of the computer you used (please write the workstation name)?

	Array Size	8K	16K	32K	64K	128K	256K
(ms)	t2-t1	1.36	3.67	7.13	63.06	110.03	224.93
	# accesses a[i]	1638400	3276800	6553600	13107200	26214400	52428800
(ns)	# mean access time	0.83	1.12	1.09	4.81	4.19	4.29

A capacidade da cache do PC usado (lab6p1) é 32K porque é notável a subida do mean access time de 32K para 64K. Isto significa que a partir da capacidade 64K, a miss rate aumenta.

Consider the data presented in Figure 1. Answer the following questions (2, 3, 4) about the machine used to generate that data.

2. What is the cache capacity?

A capacidade da cache do PC representado é 64K. Nos gráficos há uma diferença visível entre as curvas que representam os array's de tamanho 64K ou abaixo e aqueles que representam arrays de tamanho superior a 64K, esta chegando e mantendo-se em valores muito superiores de tempo, o que significa que a partir da capacidade 64K, a miss rate aumenta.

3. What is the size of each cache block?

O tamanho de cada bloco é 16B visto que as linhas do gráfico estabilizam-se a partir desse stride

4. What is the L1 cache miss penalty time?

Podemos assumir que o miss rate para array sizes acima de 64KB aproxima-se o suficiente de 100% para simplesmente assumirmos 100%. Então é dado por 9xx - 3xx = 600ns.

3 Procedure

3.1.1 Modeling the L1 Data Cache

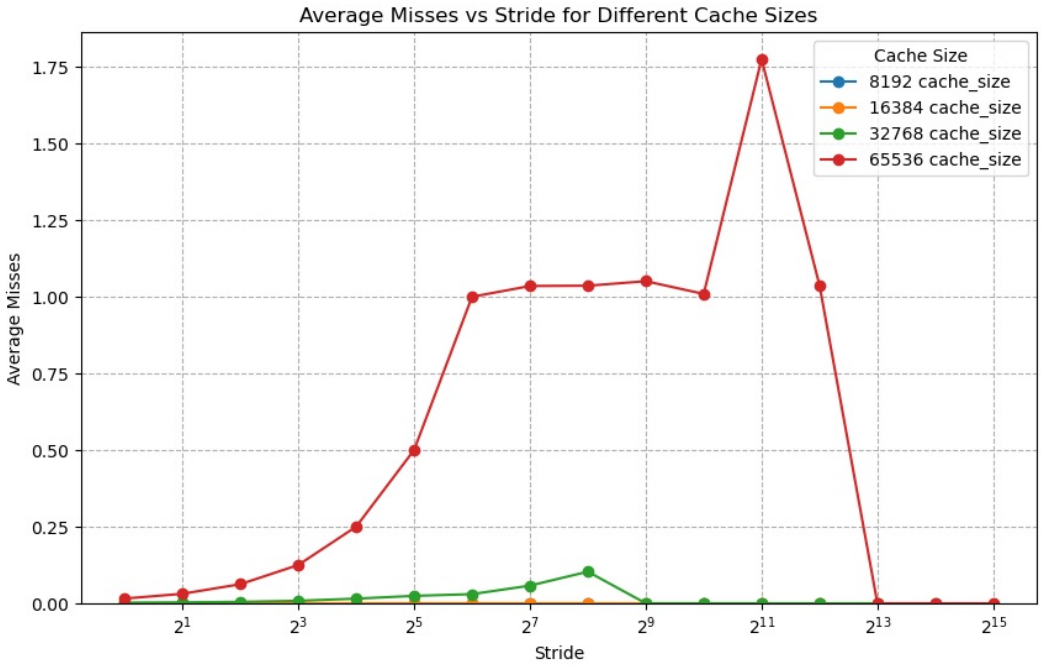
a) What are the processor events that will be analyzed during its execution? Explain their meaning.

Os eventos do processador analisados são L1 cache misses. Acontece quando a linha da cache L1 indexada pelo endereço apresenta o valid bit desligado (0) ou a Tag presente na linha não é a mesma que está no endereço. É necessário copiar um novo bloco para a cache a partir de níveis inferiores de memória (exemplo: L2 cache ou DRAM).

b) Plot the variation of the average number of misses (*Avg Misses*) with the `stride` size, for each considered dimension of the L1 data cache (8kB, 16kB, 32kB and 64kB).
Note that, you may fill these tables and graphics (as well as the following ones in this report) on your computer and submit the printed version.

Array Size	Stride	Avg Misses	Avg Cycl Time
8kBytes	1	0.000207	0.002702
	2	0.000138	0.002799
	4	0.000073	0.002776
	8	0.000062	0.002724
	16	0.000082	0.002776
	32	0.000075	0.002771
	64	0.000086	0.002249
	128	0.000038	0.002057
	256	0.000021	0.001998
	512	0.000015	0.001986
	1024	0.000010	0.001944
	2048	0.000010	0.002017
	4096	0.000013	0.002079
16kBytes	1	0.000251	0.002246
	2	0.000209	0.002242
	4	0.000267	0.002232
	8	0.000246	0.002168
	16	0.000242	0.002229
	32	0.000254	0.002235
	64	0.000181	0.002229
	128	0.000091	0.002184
	256	0.000058	0.002069
	512	0.000047	0.001997
	1024	0.000018	0.001985
	2048	0.000009	0.002081
	4096	0.000010	0.002173
	8192	0.000004	0.002078

Array Size	Stride	Avg Misses	Avg Cycl Time
32kBytes	1	0.002058	0.002220
	2	0.003100	0.002209
	4	0.003000	0.002218
	8	0.005023	0.002178
	16	0.002166	0.002166
	32	0.008907	0.002203
	64	0.012416	0.002227
	128	0.011285	0.002197
	256	0.024341	0.002187
	512	0.008361	0.002060
	1024	0.093239	0.002018
	2048	0.000025	0.002045
	4096	0.000016	0.002188
	8192	0.000018	0.002217
	16384	0.000010	0.002083
64kBytes	1	0.015150	0.001979
	2	0.031284	0.001894
	4	0.062606	0.002139
	8	0.125308	0.002233
	16	0.250528	0.002280
	32	0.500634	0.002255
	64	0.999266	0.001858
	128	1.023906	0.001876
	256	1.033866	0.001902
	512	1.043774	0.001940
	1024	0.997378	0.001905
	2048	1.002578	0.002189
	4096	1.699602	0.007282
	8192	0.000303	0.002245
	16384	0.000011	0.002175
	32768	0.000005	0.002108



2^{20}

Nota: Devemos estudar o tamanho de array superior à cache de forma a que a array não caiba na cache e como consequência provoca que o nº de avg-misses suba e assim pode-os estudar os limites da cache.

Papel A.

c) By analyzing the obtained results:

- Determine the **size** of the L1 data cache. Justify your answer.

A capacidade da cache do PC representado é 32K. Nos gráficos há uma diferença visível entre as curvas que representam os array's de tamanho 32 K ou abaixo e aqueles que representam arrays de tamanho superior a 32 K, esta chegando e mantendo-se em valores muito superiores de tempo, o que significa que a partir da capacidade 32K, a miss rate aumenta.

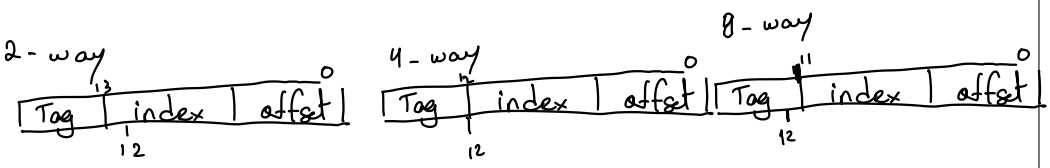
- Determine the **block size** adopted in this cache. Justify your answer.

O tamanho de cada bloco é 64B visto que as linhas do gráfico estabilizam-se a partir desse stride, o que significa que o stride é maior ou igual do que o tamanho do bloco provocando o maior nº de avg_misses. Logo o tamanho do bloco é o 1º stride onde o nº de avg-misses estabiliza (os acessos provocam-miss de forma consistente).

- Characterize the **associativity set size** adopted in this cache. Justify your answer.

Com endereçamento ao byte o offset ocupará 6 bits. Tendo em conta que o cache size é 32 kB e o block-size é 64B o nº de blocos é 2^9, ou seja uma cache direct mapped o nº de bits do index é 9. Para strides maiores que o block size em que o avg. misses é igual a 0, podemos afirmar que os endereços acedidos estão no mesmo set, logo (no endereço) o index é o mesmo.

Para o stride 2^{13} o bit nº 13 muda, pera os acessos serem no mesmo set o bit que muda tem de estar na Tag e não no index



A cache L1 é 8 way Associative

→ Para strides menores que 2^{13}: avg_misses > 0, logo não podemos assumir que estão no mesmo set

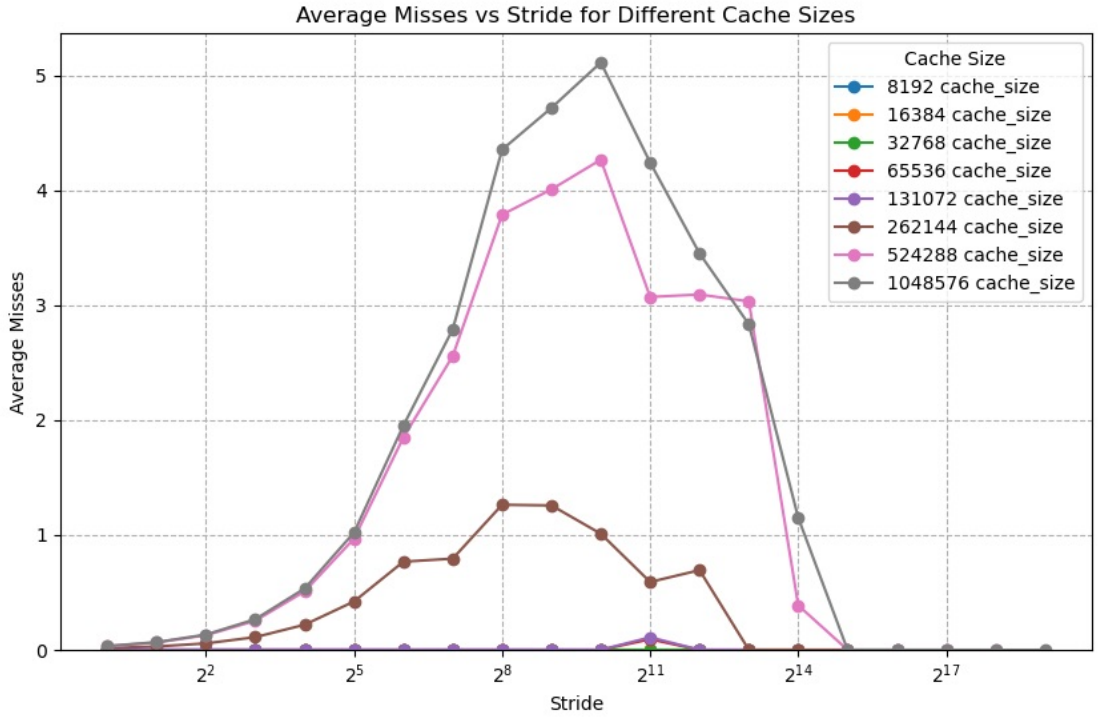
→ Para strides maiores apenas podemos afirmar que a cache é pelo menos 2-way ou 4-way Associative, já que há strides menores com avg_misses = 0

3.1.2 Modeling the L2 Cache

a) Describe and justify the changes introduced in this program.

Alteramos 2 coisas: O evento PAPI analisado deixou de ser PAPI_L1_DCM para PAPI_L2_DCM e o tamanho da cache max. A primeira deve-se à natureza do que nos foi pedido e a segunda porque L2 precisa de ser bastante maior que L1, para os misses de L1 irem para L2; o array de procura de L2 terá de ser maior. (encontra os limites de L2)

b) Plot the variation of the average number of misses (*Avg Misses*) with the `stride` size, for each considered dimension of the L2 cache.



c) By analyzing the obtained results:

- Determine the **size** of the L2 cache. Justify your answer.

A capacidade da cache do PC representado é 256K.
Nos gráficas há uma diferença visivel entre as curvas que representam os arrays de tamanho 256 K ou abaixo e aquelas que representam arrays de tamanho superior a 256K, esta chegando e mantendo-se em valores muito superiores de tempo, o que sigmifica que a partir da capacidade 256K a miss rate aumenta.

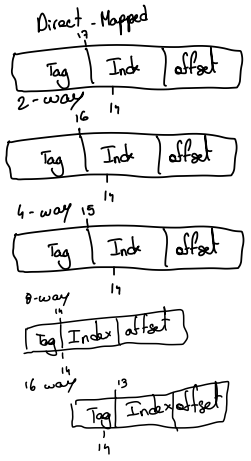
- Determine the **block size** adopted in this cache. Justify your answer.

O tamanho de cada bloco é 256 B visto que as linhas do gráfico estabilizam-se a partir desse stride, o que significa que o stride é maior ou igual do que o tamanho do bloco provocando o maior nº de avg_misses. Logo o tamanho do bloco é o 1º stride onde o nº de avg_misses estabiliza (os acessos provocam miss de forma consistente).

- Characterize the **associativity set size** adopted in this cache. Justify your answer.

Com endereçamento ao byte o offset ocupará 6 bits. Tendo em conta que o cache size é 256 KB e o block-size é 256 B o nº de blocos é 2^{10}, ou seja uma cache direct mapped o nº de bits do index é 10. Para strides maiores que o block size em que o avg. misses é igual a 0, podemos afirmar que os endereços acedidos estão no mesmo set, logo (no endereço) o index é o mesmo.

Para o stride 2^{15} o bit nº 15 muda, para os acessos serem no mesmo set o bit que muda tem de estar na Tag e não no index



A cache L2 é 16 way Associative

→ Para strides menores que 2^{15}: avg_misses > 0, logo não podemos assumir que estão no mesmo set

→ Para strides maiores apenas podemos afirmar que a cache é pelo menos 2-way ou 4-way Associative ou 8-way Associative, já que há strides menores com avg_misses = 0

3.2 Profiling and Optimizing Data Cache Accesses

3.2.1 Straightforward implementation

a) What is the total amount of memory that is required to accommodate each of these matrices?

$\frac{512 \times 512}{\text{nº de elementos}} \times \underbrace{2\,B}_{\text{espaço por elemento}} = 2^{18}\,B \times 2\,B = 2^{19}$

b) Fill the following table with the obtained data.

Total number of L1 data cache misses	135,127217	$\times 10^6$
Total number of load / store instructions completed	402.654058	$\times 10^6$
Total number of clock cycles	645.520367	$\times 10^6$
Elapsed time	0.215174	seconds

c) Evaluate the resulting L1 data cache *Hit-Rate*:

$m.r = \frac{135,127217}{402.654058} = 0,335591$

$h.r = 1 - m.r = 1 - 0,335591 = 0,664409$

3.2.2 First Optimization: Matrix transpose before multiplication [2]

a) Fill the following table with the obtained data.

Total number of L1 data cache misses	4.218790	$\times 10^6$
Total number of load / store instructions completed	402.654022	$\times 10^6$
Total number of clock cycles	548.806611	$\times 10^6$
Elapsed time	0.182935	seconds

b) Evaluate the resulting L1 data cache *Hit-Rate*:

$$h.r = \frac{402.654022 - 4.218790}{402.654022} = 0,989671$$

c) Fill the following table with the obtained data.

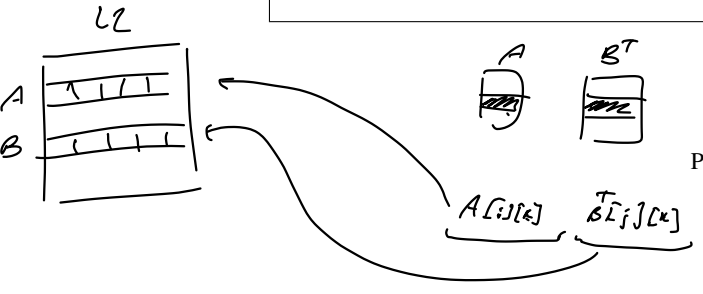
Total number of L1 data cache misses	4.482823	$\times 10^6$
Total number of load / store instructions completed	402.916293	$\times 10^6$
Total number of clock cycles	550.624153	$\times 10^6$
Elapsed time	0.183542	seconds

Comment on the obtained results when including the matrix transposition in the execution time:

A transposição da matriz não mostra afetar no tempo, tendo um efeito desprezível no elapsed time

d) Compare the obtained results with those that were obtained for the straightforward implementation, by calculating the difference of the resulting hit-rates ($\Delta\text{HitRate}$) and the obtained speedups.

$\Delta\text{HitRate} = \text{HitRate}_{\text{mm2}} - \text{HitRate}_{\text{mm1}}$: 0.989671 - 0,664509 = 0,325262
$\text{Speedup}(\#\text{Clocks}) = \#\text{Clocks}_{\text{mm1}}/\#\text{Clocks}_{\text{mm2}}$: 645.520367/548.806611 = 1,17623
$\text{Speedup}(\text{Time}) = \text{Time}_{\text{mm1}}/\text{Time}_{\text{mm2}}$: 0.215174 / 0.182935 = 1,17623
Comment: Foi obtido um speedup de ≈ 1,18 comparativamente à versão original. A hit rate aumentou 33%, isto pode ser explicado pelo aumento de L2 cache hits, já que com a matriz transposta é possível guardar na cache L2 vários elementos da mesma linha da matriz e devido à localidade espacial a hit rate aumenta.



3.2.3 Second Optimization: Blocked (tiled) matrix multiply [2]

a) How many matrix elements can be accommodated in each cache line?

$$\frac{\text{block\_size}}{\text{tam cada elemento}} = \frac{64}{2} = 32 \text{ elementos}$$

b) Fill the following table with the obtained data.

Total number of L1 data cache misses	2.601769	$\times 10^6$
Total number of load / store instructions completed	403.317487	$\times 10^6$
Total number of clock cycles	222.831276	$\times 10^6$
Elapsed time	0.074278	seconds

c) Evaluate the resulting L1 data cache *Hit-Rate*:

$$\text{Hit Rate} = \frac{403.317487 - 2.601769}{403.317487} = 0.993549$$

d) Compare the obtained results with those that were obtained for the straightforward implementation, by calculating the difference of the resulting hit-rates ($\Delta\text{HitRate}$) and the obtained speedup.

$\Delta\text{HitRate} = \text{HitRate}_{\text{mm3}} - \text{HitRate}_{\text{mm1}}$: 0.993549 − 0.664409 = 0.32914
$\text{Speedup}(\#\text{Clocks}) = \#\text{Clocks}_{\text{mm1}}/\#\text{Clocks}_{\text{mm3}}$: 645.520367/222.831276 = 2.8969
Comment: O Hit Rate aumentou cerca de 33% e houve um speedup de cerca de 2.90 ao fazer a otimização presente em mmm3. A matriz foi dividida em sub matrizes cujo tamanho foi calculado para caber num bloco da cache L1 (localidade espacial) uma melhor abordagem do que mmm1 que resulta em acessos na 2ª matriz a posições de memória muito separadas, ou seja, não cabe num bloco que significa mais cache misses. Resultando num aumento de hit rate de 33%..

e) Compare the obtained results with those that were obtained for the matrix transpose implementation by calculating the difference of the resulting hit-rates ($\Delta\text{HitRate}$) and the obtained speedup. If the obtained speedup is positive, but the difference of the resulting hit-rates is negative, how do you explain the performance improvement? (Hint: study the hit-rates of the L2 cache for both implementations;)

$\Delta\text{HitRate} = \text{HitRate}_{\text{mm3}_{L1}} - \text{HitRate}_{\text{mm2}_{L1}}$: 0.993549 - 0.996671 = -0.003122
$\text{Speedup}(\#\text{Clocks}) = \#\text{Clocks}_{\text{mm2}} / \#\text{Clocks}_{\text{mm3}}$: 548.806611 / 222.831276 = 2,462 87963
Comment: Apesar da Δ Hit Rate ser negativa ainda podemos observar um speedup considerável, uma vez que apesar de o programa mm3 apresentar uma hit rate ligeiramente menor, a percentagem de acessos de L2 que resultaram no miss é bastante inferior à percentagem de acessos a L2 que resultaram num miss no programa mm2. Devido a isto a miss penalty diminui, reduzindo o tempo de acessos à memória no programa mm3, que é inferior ao de mm2, explicando o speedup obtido

3.2.3 Comparing results against the CPU specifications

Now that you have characterized the cache on your lab computer, you are going to compare it against the manufacturer's specification. For this you can check the device's datasheet, or make use of the command `lscpu`. Comment the results.

De acordo com o comando: L1 tem 32KB de cache size L2: 256KB de cache size.
Isto está correto de acordo com os nossos cálculos
(para cada core)

A PAPI - Performance Application Programming Interface

The PAPI project [1] specifies a standard Application Programming Interface (API) for accessing hardware performance counters available in most modern microprocessors. These counters exist as a small set of registers that count *Events*, defined as occurrences of specific signals related to the processor's function (such as cache misses and floating point operations), while the program executes on the processor. Monitoring these events may have a variety of uses in the performance analysis and tuning of an application, since it facilitates the correlation between the source/object code structure and the efficiency of the actual mapping of such code to the underlying architecture. Besides performance analysis, and hand tuning, this information may also be used in compiler optimization, debugging, benchmarking, monitoring and performance modeling.

PAPI has been implemented on a number of different platforms, including: Alpha; MIPS R10K and R12K; AMD Athlon and Opteron; Intel Pentium II, Pentium III, Pentium M, Pentium IV, Itanium 1 and Itanium 2; IBM Power 3, 4 and 5; Cell; Sun UltraSparc I, II and II, etc.

Although each processor has a number of events that are native to that specific architecture, PAPI provides a software abstraction of these architecture-dependent *Native Events* into a collection of *Preset Events*, also known as *predefined events*, that define a common set of events deemed relevant and useful for application performance tuning. These events are typically found in many CPUs that provide performance counters. They give access to the memory hierarchy, cache coherence protocol events, cycle and instruction counts, functional unit, and pipeline status. Hence, preset events may be regarded as mappings from symbolic names (PAPI preset name) to machine specific definitions (native countable events) for a particular hardware resource. For example, Total Cycles (in user mode) is mapped into PAPI_TOT_CYC. Some presets are derived from the underlying hardware metrics. For example, Total L1 Cache Misses (PAPI_L1_TCM) is the sum of L1 Data Misses and L1 Instruction Misses on a given platform. The list of preset and native events that are available on a specific platform can be obtained by running the commands `papi_avail` and `papi_native_avail`, both provided by the papi source distribution.

Besides the standard set of events for application performance tuning, the PAPI specification also includes both a high-level and a low-level sets of routines for accessing the counters. The high level interface consists of eight functions that make it easy to get started with PAPI, by simply providing the ability to start, stop, and read sets of events. This interface is intended for the acquisition of simple but accurate measurement by application engineers [3, 4]:

- `PAPI_num_counters` – get the number of hardware counters available on the system;
- `PAPI_flops` – simplified call to get Mflops/s (floating point operation rate), real and processor time;
- `PAPI_ipc` – gets instructions per cycle, real and processor time;
- `PAPI_accum_counters` – add current counts to array and reset counters;
- `PAPI_read_counters` – copy current counts to array and reset counters;
- `PAPI_start_counters` – start counting hardware events;
- `PAPI_stop_counters` – stop counters and return current counts.

The following is a simple code example of using the high-level API [3, 4]:

```
#include <papi.h>

#define NUM_FLOPS 10000
#define NUM_EVENTS 1

int 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);

  do_some_work();

  /* 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_some_work();

  /* Add the counters */
  if (PAPI_accum_counters(values, NUM_EVENTS) != PAPI_OK)
    handle_error(1);

  printf("After adding the counters: %lld\n", values[0]);

  do_some_work();

  /* Stop counting events */
  if (PAPI_stop_counters(values, NUM_EVENTS) != PAPI_OK)
    handle_error(1);

  printf("After stopping the counters: %lld\n", values[0]);
}
```

Possible output:

```
After reading the counters: 441027
After adding the counters: 891959
After stopping the counters: 443994
```

The fully programmable low-level interface provides more sophisticated options for controlling the counters, such as setting thresholds for interrupt on overflow, as well as access to all native counting modes and events. Such interface is intended for third-party tool writers or users with more sophisticated needs.

The PAPI specification also provides access to the most accurate timers available on the platform in use. These timers can be used to obtain both real and virtual time on each supported platform: the real time clock runs all the time (e.g., a wall clock), while the virtual time clock runs only when the processor is running in user mode.

In the following code example, `PAPI_get_real_cyc()` and `PAPI_get_real_usec()` are used to obtain the real time it takes to create an event set in clock cycles and in microseconds, respectively [3, 4]:

```
#include <papi.h>

int main(){
  long long start_cycles, end_cycles, start_usec, end_usec;
  int EventSet = PAPI_NULL;

  if (PAPI_library_init(PAPI_VER_CURRENT) != PAPI_VER_CURRENT)
    exit(1);

  /*Create an EventSet */
  if (PAPI_create_eventset(&EventSet) != PAPI_OK)
    exit(1);

  /* Gets the starting time in clock cycles */
  start_cycles = PAPI_get_real_cyc();

  /* Gets the starting time in microseconds */
  start_usec = PAPI_get_real_usec();

  do_some_work();

  /* Gets the ending time in clock cycles */
  end_cycles = PAPI_get_real_cyc();

  /* Gets the ending time in microseconds */
  end_usec = PAPI_get_real_usec();

  printf("Wall clock cycles: %lld\n", end_cycles - start_cycles);
  prinf("Wall clock time in microseconds: %lld\n", end_usec - start_usec);
}
```

Possible output:

```
Wall clock cycles: 100173
Wall clock time in microseconds: 136
```