First Lab Assignment: System Modeling and Profiling

STUDENTS IDENTIFICATION:

Number:	Name:

2 Exercise

Please justify all your answers with values from the experiments.

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

Array Size	4 KB	8 KB	16 KB	32 KB	64 KB	128 KB
t2-t1 (s)	0,000728	0,001505	0,003097	0,006158	0,032595	0,051371
# accesses a[i]	819200	1638400	3276800	6553600	13107200	26214400
# mean access time (ns)	0,887	0,919	0,945	0,940	2,486	1,960

Através da execução do código spark.c no PC 5 do lab3 (lab3p5) constatamos os valores na tabela ao escolher o stride 2048. Analisando os valores de "#mean access time" observamos um salto significativo de 32KiB a 64KiB logo o valor da cache é 32KiB, depois desse valor haverão demasiados misses.

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?

Através da observação do gráfico, conseguimos ver que de 64K a 128K há um grande salto no tempo o que indica que de 4K a 64K há maioritariamente hits e de 64K a 4M maioritariamente misses. Logo a capacidade da cache é 64Kbytes. Este salto indica que o tamanho do array é superior à capacidade da cache.

3. What is the size of each cache block?

Se a cache capacity é 64KB, pelas razões indicadas acima, então observando o comportamento no gráfico para o array de tamanho 64KB, constatamos que depois do stride 16 há uma estabilização, logo o tamanho de cada cache block é de 16 bytes.

4. What is the L1 cache miss penalty time?

$t\_hit_{(ns)} \simeq 360$ (através da observação do gráfico)
$t\_miss_{(ns)} \simeq 1000$ (através da observação do gráfico)

cache miss penalty = t_miss - t_hit = 1000 - 360
= 640 ns

3 Procedure

3.1.1 Modeling the L1 Data Cache

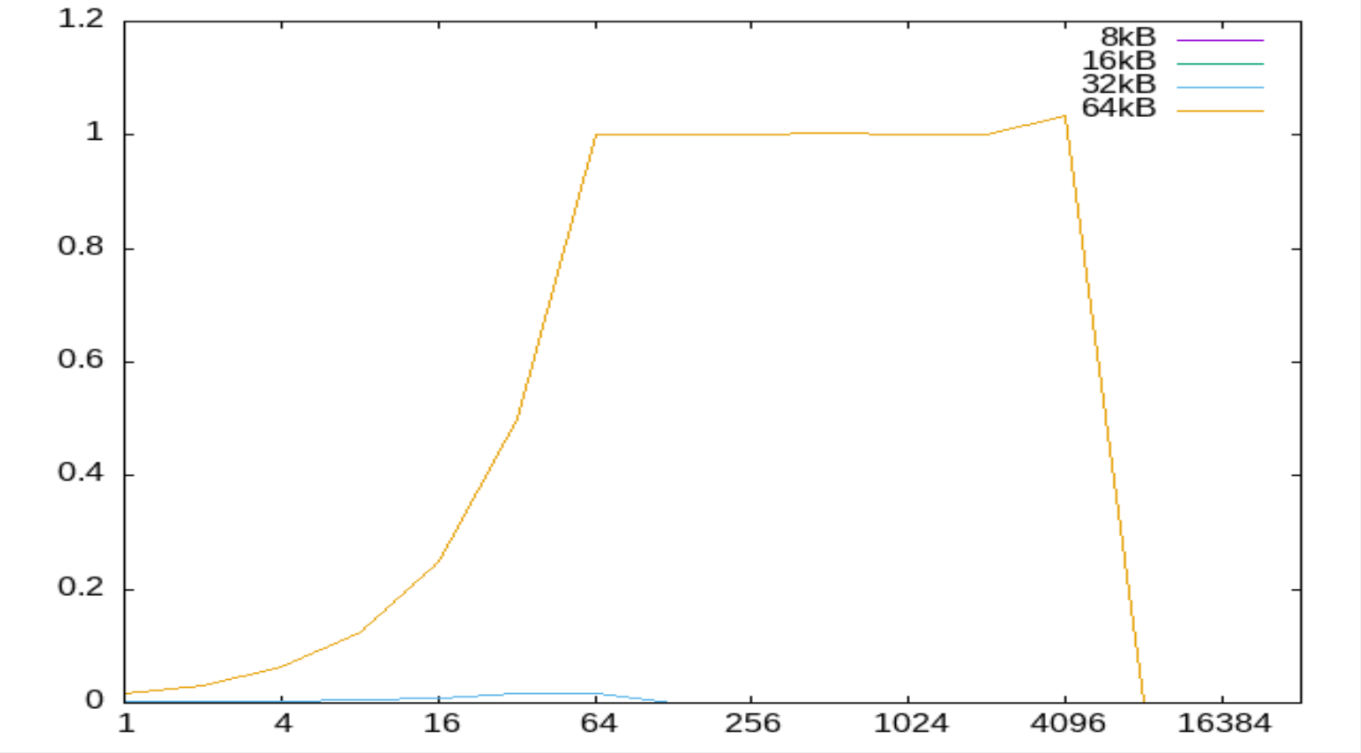
a) What are the processor events that will be analyzed during its execution? Explain their meaning.

Os eventos do processador analisados são os average cache misses por tamanho de stride para cada tamanho da variável cache_size, ou seja, consoante o tamanho do salto que se executa, avalia-se para cada cache_size quantas vezes ocorrem caches misses em média por ciclo através do PAPI que adiciona a métrica PAPI_L1_DCM aos eventos por si analisados; além disso, analisa-se ainda o tempo de execução para cada stride, para cada tamanho da variável cache_size, através dos ciclos de relógio contabilizados.

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	0,000213	0,00 2141
	2	0,000172	0,00 2143
	4	0,000078	0,00 2135
	8	0,000048	0,00 2062
	16	0,000040	0,00 2020
	32	0,000045	0,00 1980
	64	0,000041	0,00 1994
	128	0,000028	0,00 1926
	256	0,000016	0,00 2003
	512	0,000015	0,001780
	1024	0,000009	0,00 1782
	2048	0,000010	0,00 1829
	4096	0,000007	0,00 1906
16kBytes	1	0,000160	0,00 2052
	2	0,000124	0,00 2058
	4	0,000153	0,00 2060
	8	0,000128	0,00 2028
	16	0,000149	0,00 2069
	32	0,000144	0,00 2035
	64	0,000140	0,00 2016
	128	0,000082	0,00 2002
	256	0,000041	0,00 1967
	512	0,000023	0,00 2041
	1024	0,000015	0,00 1812
	2048	0,000006	0,00 1891
	4096	0,000005	0,00 1978
	8192	0,000005	0,00 1902

Array Size	Stride	Avg Misses	Avg Cycl Time
32kBytes	1	0,001329	0,001990
	2	0,001627	0,001991
	4	0,003345	0,001994
	8	0,007380	0,001954
	16	0,013454	0,001904
	32	0,025529	0,001906
	64	0,032337	0,002007
	128	0,066756	0,001991
	256	0,000261	0,001932
	512	0,000161	0,001882
	1024	0,000087	0,001992
	2048	0,000032	0,001816
	4096	0,000016	0,001952
	8192	0,000004	0,001940
	16384	0,000003	0,001861
64kBytes	1	0,015650	0,001798
	2	0,031277	0,001797
	4	0,062666	0,001973
	8	0,125272	0,002002
	16	0,250693	0,001983
	32	0,501467	0,002015
	64	1,000768	0,001654
	128	1,001182	0,001681
	256	1,002165	0,001656
	512	1,004424	0,001681
	1024	1,000003	0,001727
	2048	1,031697	0,002190
	4096	1,031710	0,004496
	8192	0,000094	0,001987
	16384	0,000001	0,001936
	32768	0,000021	0,001862



c) By analyzing the obtained results:

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

Como se pode observar no gráfico produzido, as average misses por ciclo (eixo y) aumentam substancialmente quando um tamanho de cache_size de 64KB. Logo, o tamanho adequado de cache L1 é 32KB, uma vez que é o maior tamanho logo abaixo daquele em que existe um aumento substancial de misses por ciclo, e ele próprio tem uma baixíssima quantidade de misses por ciclo em média segundo o gráfico.

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

Com base no gráfico, observamos que para o tamanho de cache_size de 64KB, existe um "plateau" de average misses por ciclo que se inicia no stride 64. Isto sugere que este stride irá aproximar-se do tamanho de bloco adotado na cache L1, uma vez que, em cada deslocamento, lê um novo bloco, resultando em misses consistentes ao ao ler novos blocos. Então, conclui-se que o tamanho de bloco desta cache é 64B.

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

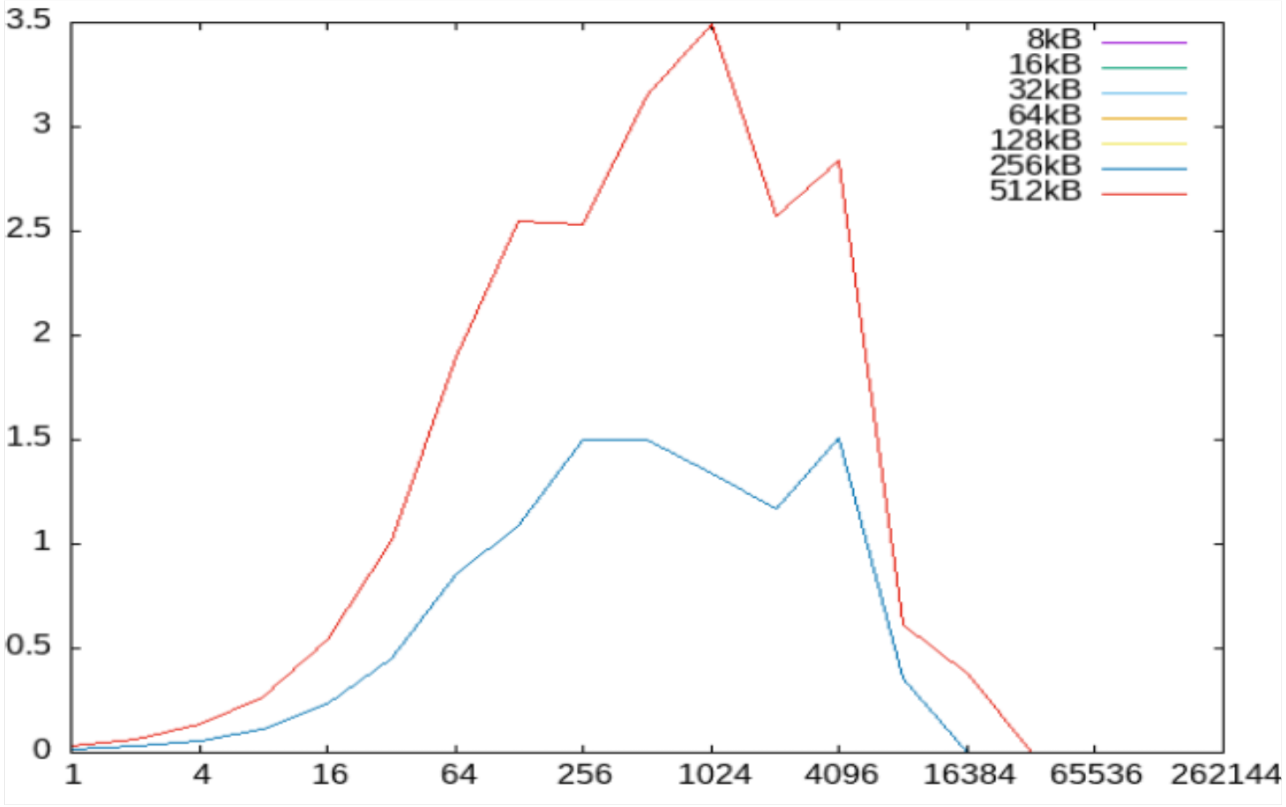
Analisando o gráfico, observa-se uma queda abrupta de average misses por ciclo para o array-size de 64KB no stride 8096 para zero average misses por ciclo. Conclui-se que o tamanho da cache L1 é 32KB; para os endereçar, necessitamos de 15 bits (2^{15} = 32768); determinámos ainda que o tamanho do bloco é 64B, portanto necessitamos de 6 bits para representar o offset (2^6 = 64). Logo, temos 15(total) - 6(offset) = 9 bits disponíveis no máximo para representar o índice. Uma vez que o tamanho de array em que se observa a queda é de 64KB, e ocorreu no stride 8192, existem (64 × 1024)/8192 = 8 acessos iniciando em 0, com saltos de 8192 até 57344); analisando as associatividades possíveis, verificamos que tanto mapeamento direto como 2 way e 4 way associative set size irão causar conflitos, uma vez que endereços com tags diferentes acedem ao mesmo bloco. Para 8 way associative set size, conseguem-se alcançar todos os acessos sem se provocarem misses apesar dos endereços terem tags diferentes. Se considerássemos a possibilidade de ser 16 way associative set size, para um caso em que existissem 16 acessos ao array não existiriam misses também (stride de 4096), o que se pode observar pelo gráfico que não acontece, uma vez que para um stride de 4096 no array de 64KB a average miss rate por ciclo é próxima de 1. Logo, qualquer associativity set size igual ou superior não é adequado, e conclui-se que o associativity set size utilizado nesta cache é 8 way associativity set size.

3.1.2 Modeling the L2 Cache

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

Alterou-se a instrução PAPI_add_event(EventSet, PAPI_L1_DCM) para PAPI_add_event(EventSet, PAPI_L2_DCM) a fim de analisar as cache misses da L2, a métrica relevante para saber o tamanho da cache e o block size. Além disso, como a constante CACHE_MAX (que estava a 64KB) não permitia concluir o tamanho da cache utilizado, procedemos a aumentá-lo para 512KB, o que produziu um gráfico a partir do qual já podíamos concluir algo.

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.

Pelo gráfico produzido e seguindo o mesmo raciocínio da cache L1, observa-se um aumento substancial das average misses por ciclo para um tamanho de 512 KB. Postanto, o tamanho mais adequado de cache será 256 KB uma vez que é o maior tamanho testado para o qual existe um valor baixo de average misses por ciclo, e que se mantém baixo ao longo do aumento do stride.

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

Com base no gráfico produzido, é possível observar para o maior tamanho testado (512KB) que existe um "plateau" de average misses por ciclo no stride 128. Pela mesma lógica explicada para a cache L1, prevê-se que este se aproxime do tamanho de bloco utilizado na cache L2, uma vez que em cada deslocamento deste irá ler um bloco novo, resultando em misses consistentes. Logo, o tamanho de bloco adotado nesta cache é 128 B, pelas razões apresentadas.

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

Seguindo o mesmo raciocínio que a Cache L1, observa-se no gráfico, para o maior tamanho de cache_size testado (512KB) que as average misses por cada teste tamanho de cache_size atingem 0 no stride 32768. Conclui-se anteriormente que o tamanho da cache L2 é 256KB, para os endereços, necessitamos de 18 bits (2^{18} = 262144); determinámos ainda que o tamanho do bloco é 128B, portanto necessitamos de 7 bits para representar o offset (2^7 = 128). Logo, temos 18(total) - 7(offset) = 11 bits disponíveis no máximo para representar o índice. Uma vez que o tamanho de array em que se observa a queda é de 512KB e ocorreu no stride 32768, existem (512x1024)/32768 = 16 acessos (iniciando em 0, com saltos de 32768 até 524288). Pelo mesmo raciocínio que na cache L1, verifica-se que mapeamento direto e 2way, 4way e 8way associativity não causam misses, portanto não podem ser essas as utilizadas.
Para 16 way associative setsize, consegue-se acomodar todos os 16 acessos sem se provocarem misses apesar dos endereços terem tags diferentes. Para eliminar a possibilidade de serem associativity set sizes superiores, se considerarmos 32 way associative set size, se existissem 32 acessos para o tamanho de cache_size testado (512KB) não existiriam conflitos, logo não haveriam misses; isto ocorreria no stride 16384. Porem, pode-se constatar que isto não acontece uma vez que o gráfico apresenta um valor superior a 0 para um tamanho de cache_size de 512KB, no stride 16384 (ligeiramente inferior a 0.5).
Logo, qualquer associativity setsize igual ou superior a 32 way não é adequado, e conclui-se que o associativity setsize utilizado nesta cache é 16way associativity set size.

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?

Cada uma das 3 matrizes (mul1, mul2, res) são do tamanho: N x N e N está definido no código como N = 512. Logo para descobrir o tamanho de cada matriz é necessário: N x N = 262144 bytes e além disso temos de ter em atenção o tipo de cada matriz (int16_t) que é 2bytes. (int16_t → 16bits)
Logo é necessário alocar para cada matriz 262144 x 2 = 524288 bytes em memória.

b) Fill the following table with the obtained data.

Total number of L1 data cache misses	135,040585	$\times 10^6$
Total number of load / store instructions completed	536,871337	$\times 10^6$
Total number of clock cycles	565,031594	$\times 10^6$
Elapsed time	0,182500	seconds

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

L1 data cache Hit-Rate (%): $\frac{\text{L1 Data cache Hits}}{\text{number load instructions}} \times 100 = \frac{402653286 - 135040585}{402653286} \times 100$
(load inst. - misses)
$\simeq 66{,}46\,\%$

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	4,216507	$\times 10^6$
Total number of load / store instructions completed	536,197497	$\times 10^6$
Total number of clock cycles	486,976834	$\times 10^6$
Elapsed time	0,157034	seconds

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

L1 data cache hit-rate = $\frac{\text{L1 Data cache hits}}{\text{number load inst.}} \times 100 = \frac{\text{load inst.} - \text{L1\_DCM}}{\text{load ins.}} \times 100$

$= \frac{401979446 - 4216507}{401979446} \times 100 \simeq 98,95\%$

c) Fill the following table with the obtained data.

Total number of L1 data cache misses	4,480143	$\times 10^6$
Total number of load / store instructions completed	537,395803	$\times 10^6$
Total number of clock cycles	476,946927	$\times 10^6$
Elapsed time	0,154049	seconds

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

Ao transpor a segunda matriz antes de efetuar a multiplicação, observa-se que o tempo de execução diminuiu de 0,157034 s para 0,154049 s; apesar da otimização se traduzir numa diminuição pouco significativa do tempo, para matrizes de grande dimensão pode fazer uma diferença substancial.

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.

$\Delta\text{HitRate} = \text{HitRate}_{\text{mm2}} - \text{HitRate}_{\text{mm1}}$: 98,95 - 66,46 = 32,49 %
$\text{Speedup}(\#\text{Clocks}) = \#\text{Clocks}_{\text{mm1}}/\#\text{Clocks}_{\text{mm2}}$: $565,031594 \times 10^6 / 476,946927 \times 10^6 = 1.18$
$\text{Speedup}(\text{Time}) = \text{Time}_{\text{mm1}}/\text{Time}_{\text{mm2}}$: 0,182506/0,154049 ≃ 1.18
Comment: Com a transposição da segunda matriz antes da multiplicação, observa-se que a hit rate sobe substancialmente (32,49%), e que existe um ligeiro speedup de 1,18, melhorando o tempo de execução do programa.

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

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

cache_line_size = 64 B

$$n^{o}\ elementos = \frac{64}{2B\ por\ elemento} = 32\ elementos\ por\ linha$$

b) Fill the following table with the obtained data.

Total number of L1 data cache misses	4,732918 $\times 10^6$
Total number of load / store instructions completed	537,802089 $\times 10^6$
Total number of clock cycles	272,161336 $\times 10^6$
Elapsed time	0,087905 seconds

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

$$Hit\ rate = \frac{L1\ data\ cache\ hits}{number\ load\ inst.} \times 100 = \frac{load\ inst. - L1\_DCM}{load\ inst.} \times 100$$

$$= \frac{403,182353 - 4,732918}{403,182353} \times 100 \simeq 98,83$$

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

$\Delta\mathrm{HitRate} = \mathrm{HitRate}_{\mathrm{mm3}} - \mathrm{HitRate}_{\mathrm{mm1}}$: 98,83 - 66,46 = 32,37 %
$\mathrm{Speedup}(\#\mathrm{Clocks}) = \#\mathrm{Clocks}_{\mathrm{mm1}}/\#\mathrm{Clocks}_{\mathrm{mm3}}$: 565031594/272161336 = 2,077
Comment: A variação do hit rate e o speedup, são positivos, mostrando que é mais eficiente multiplicar matrizes com o mm3.

e) Compare the obtained results with those that were obtained for the matrix transpose implementation by calculating the difference of the resulting hit-rates ($\Delta\mathrm{HitRate}$) 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;)

$\Delta\text{HitRate} = \text{HitRate}_{\text{mm3}} - \text{HitRate}_{\text{mm2}}$: 98,83 - 98,95 = -0,12
$\text{Speedup}(\#\text{Clocks}) = \#\text{Clocks}_{\text{mm2}} / \#\text{Clocks}_{\text{mm3}}$: 476,946927 x10^6 / 272,161336 x10^6 = 1,752
Comment: Como o speedup é positivo então o tempo de execução melhorou assim como a performance, no entanto, estão a ocorrer mais cache misses e então o hit rate diminui. Avaliando os resultados obtidos ao usar a cache L2 percebemos que é maior que L1 e os misses no L1 têm um impacto mais imediato na eficiência do programa que os da L1.

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.

Utilizando o comando lscpu no computador para o qual analisámos as caches L1 e L2 (lab3g5), verificamos que a soma de todas as caches L1 é 192KB, e existem 6 instâncias das mesmas. Logo, o tamanho de uma cache individual L1 será 192/6 = 32 KB, o que está de acordo com o tamanho que indicámos na secção 3.1.1. para o size of L1 data cache (32KB). Indica-se ainda que a soma de todas as caches L2 corresponde a 1.5MB e existem também 6 instâncias das mesmas, portanto o tamanho de uma cache L2 individual será 1536/6 = 256 KB, o que também está de acordo com o tamanho por nós indicado na secção 3.1.2 para o size of L2 data cache (256KB).
Conclui-se que a previsão teórica com base na análise dos gráficos produzidos vai de encontro às especificações do computador para as caches L1 e L2.

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
```