Performance modeling for power consumption reduction on SCC

Bertrand Putigny, Brice Goglin, Denis Barthou

Performance modeling for power consumption reduction on SCC

Bertrand Putigny[1][2], Brice Goglin[1][2], Denis Barthou[2],
[1] Inria
[2] Univ. Bordeaux, LaBRI, UMR 5800, F-33400 Talence, France

***Abstract*—As power is becoming one of the biggest challenge in high performance computing, we are proposing a performance model on the Single-chip Cloud Computer in order to predict both power consumption and runtime of regular codes. This model takes into account the frequency at which the cores of the SCC chip operate. Thus, we can predict the execution time and power needed to run the code for each available frequency. This allows to choose the best frequency to optimize several metrics such as power efficiency or minimizing power consumption, based on the needs of the application. Our model only needs some parameters that are code dependent. These parameters can be found through static code analysis. We validated our model by showing that it can predict performance and find the optimal frequency divisor to optimize energy efficiency on several dense linear algebra codes.**

***Index Terms*—Intel SCC, performance model, performance prediction, power, energy efficiency, optimization.**

I. Introduction

Reducing power consumption is one of the main challenge in the HPC community. Indeed power is the leading design constraint for next generation of supercomputers [4]. Therefore energy efficiency is becoming an important metric to evaluate both hardware and software.

The Intel Single-chip Cloud Computer (SCC) is a good example of next generation hardware with an easy way to control power consumption. It provides a software API to control core voltage and core frequency. This opens promising opportunities to optimize power consumption and to explore new trade-offs between power and performance.

This paper aims at exploring the opportunities offered by SCC to reduce power consumption with a small impact on performance. It is organized as follow: Section II describes the model used to predict performance, the Section III demonstrates the reliability of our model by applying it to several basic linear codes, we will also explain how to choose the frequency to optimize a given metric. Sections IV, V, and VI present respectively the related work, future work and conclusion.

II. Performance model

In this section we provide a performance model in order to predict the impact of core frequency scaling on the execution time of several basic linear algebra kernels on the SCC chip.

Project ProHMPT is funded by the French National Agency for Research under the ANR-08-COSI-013 grant.

As we focus on dense linear algebra, we only need a few data to predict a given code performance. The considered datasets being too large to fit in cache, we need the execution time of one iteration of the innermost loop of the kernel and the memory latency.

A. Memory model

To build the memory model, we assume that the application can exploit perfectly data reuse and therefore we assume that each data is accessed only once. We do not take the number of cache accesses into account in the prediction of the overall memory access time because they are not actual memory accesses since the request does not have to go all the way to DRAM. Moreover the cache is not coherent. Therefore there is no overhead due to the cache coherence protocol.

On SCC, a memory access takes 40 core cycles + $4\times n\times$ 2 mesh cycles + 46 memory cycles (DDR3 latency) where n is the number of hops between the requesting core and the memory controller [1]. In our case, we are only running sequential code, therefore we are assuming that the memory access time is $40\times c+46\times m$ cycles, where c is the number of core cycles and m the number of memory cycles. Accessing memory takes 40 core cycles plus 46 memory cycles.

Frequency scaling only affects core frequency, the memory frequency is a constant, (in our case 800MHz). Therefore, changing frequency mostly impacts the code performance if it is computation bound. The number of core cycles to perform one DDR3 access is: $40 + 46 \times \frac{core\_freq}{800}$.

As we can see from the formula dividing the core frequency by 8 (from 800MHz to 100MHz) will only reduce the memory performance by 46%

As the P54C core used in the SCC supports two pending memory requests, we can assume that accessing x elements will take $\frac{x}{2}(40 + 46 \times \frac{core\_freq}{800})$ core cycles.

B. Computational model

In order to predict the number of cycles needed to perform the computation itself we need the latency of each instruction. Agner Fog measured the latency of each x86 and x87 instruction [7]. We used his work to predict the number of cycles to perform one iteration of the innermost loops of each studied kernel. The computation model is very simple, as most of the instructions use the same execution port, there is almost no instruction parallelism. A more complex performance model, considering also measured latencies as a building block of the

Freq divisor	Tile freq (MHz)	Voltage (volts)
2	800	1.1
3	533	0.8
4	400	0.7
5	320	0.6
6	266	0.6
7	228	0.6
8	200	0.6
9	178	0.6
10	160	0.6
11	145	0.6
12	133	0.6
13	123	0.6
14	114	0.6
15	106	0.6
16	100	0.6

TABLE I: Relation between voltage and frequency.

model, is used in the performance tuning tool MAQAO [2]. We use such tool to measure the execution time of one iteration of the innermost loop. As most of the execution time of the codes we consider is spent in inner loops, this performance estimation is expected to be rather accurate.

From this computation model the impact of frequency scaling on the computation performance is straightforward. The number of cycles to perform the computation is not affected by the frequency. Thus, reducing the core frequency by a factor of x will multiply the running time by x.

C. Power model

We use a very simple power model to estimate the power saved by reducing the core frequency. Table I shows the voltage used by the tile for each frequency, these data are provided by the SCC Programmer's guide [1].

The power consumption model used in this paper is the general model:

$$P = CV^2 f$$

where C is a constant, V the voltage and f the frequency of the core. As shown In Table I the voltage is a function of the frequency, thus, we can express the power consumption as a function of the core frequency only.

We choose not to introduce a power model for the memory for two reasons: first we have no software control on the memory frequency at runtime. We can change the memory frequency by re-initializing the SCC platform but not at runtime. Thus, the memory energy consumption is constant and we have no control over it. Therefore it would be almost worthless to complicate our model with such information. The other reason is that until now we used models that can be transposed to other architectures. As the memory architecture of the SCC is very different from more general purpose architecture, its energy model would not fit for those architectures. Thus, the model described in this paper is completely general and can be easily transposed to other architectures.

D. Overall model

In this section we describe how to use both the memory and computational models to predict the performance of a given code.

As the P54C core can execute instructions while some memory requests are pending, we assume that the execution time will be the maximum between the computation time and the memory access time:

$$runtime(f_c) = MAX\left(\frac{computation}{f_c}, mem\_access(f_c)\right)$$

with f_c the core frequency.

With this runtime prediction, we estimate how a code execution is affected by changing the core frequency. Taking the decision to reduce the core frequency in order to save energy can be done with a static code analysis.

As show in Section II-A the memory access performance is almost not affected by reducing core frequency, while reducing core frequency increases dramatically the computation time. From this observation we see that reducing core frequency for memory bound code is highly beneficial for power consumption because it will almost not affect performance while reducing dramatically energy consumption. However, reducing core frequency for compute bound code will directly affect performance.

III. Model evaluation

In this section we compare our model with the real runtime of several regular codes in order to check its validity. We used three computation kernels, one BLAS-1, one BLAS-2 and one BLAS-3 kernels namely dot product, matrix-vector product and matrix-matrix product.

First let us describe how we applied our model to these three kernels: In the following formulas, f_{div} denotes the core frequency divisor (as shown in Table I) and $power(f_{div})$ the power used by the core when running at the frequency corresponding to f_{div} (see Table I). An important point is that we used large data sets that do not fit in cache so as to measure the execution time of the code. Thus, the kernel actually gets data from DRAM and not from caches. However, the matrix-matrix multiplication is tiled in order to benefit from data reuse in cache.

A. Dot product multiplication

For the dot product kernel, the memory access time in cycles is:

$$cycles_{mem}(f_{div}) = size \times \left(40 + 46 \times \frac{2}{f_{div}}\right)$$

The computation time in cycles is given by:

$$cycles_{comp}(f_{div}) = size \times \left(\frac{body}{unroll}\right),$$

with $body$ the execution time (in cycles) of the innermost loop body and $unroll$ the unroll factor of the innermost loop. In the case shown on Figure 1 $body = 36$ and $unroll = 4$. Then the power efficiency is:

$$power_{eff}(f_{div}) = \frac{flop}{\frac{model(f_{div})}{freq} \times power(f_{div})},$$

with $flop$ the number of floating point operations of the kernel, $model(f_{div})$ the number of cycles predicted by our model and $freq$ the actual core frequency ($\frac{1600}{f_{div}}$). In the case shown on Figure 1,

$$model(f_{div}) = MAX\left(cycles_{mem}(f_{div}), cycles_{comp}(f_{div})\right) = cycles_{mem}(f_{div})$$

Figure 1a shows that the number of cycles for both the memory model and obtained through benchmark decreases when frequency decreases. The reason is that frequency scaling only affects core frequency. For memory bound codes such as dot product, reducing the core frequency reduces the time spent in waiting for memory requests. However, the code is not executing faster, as shown in Figure 1b.

B. Matrix-vector product

Similarly the model for the matrix-vector product is:

$$cycles_{mem}(f_{div}) = \frac{matrix\_size}{2} \times \left(40 + 46 \times \frac{2}{f_{div}}\right)$$

$$cycles_{comp}(f_{div}) = matrix\_size \times \left(\frac{body}{unroll}\right)$$

With $matrix\_size = 512 \times 1024$ elements, $body = 64$ cycles, and $unroll = 4$ for the case shown on Figure 2.

$$power_{eff}(f_{div}) = \frac{flop}{\frac{model(f_{div})}{freq} \times power(f_{div})}$$

In this case, again, the memory access time is more important than the time for the computation, thus, the runtime is given by the memory access time. (ie. $model(f_{div}) = cycles_{mem}(f_{div})$)

Figure 2a shows that the number of cycles for both the memory model and obtained through benchmark decreases when frequency decreases. The reason is the same as for the dot product.

C. Matrix-matrix product

The model for the matrix-matrix multiplication is:

$$cycles_{mem}(f_{div}) = 3 \times \frac{matrix\_size^2}{2} \times \left(40 + 46 \times \frac{2}{f_{div}}\right)$$

$$cycles_{comp}(f_{div}) = matrix\_size^3 \times \left(\frac{body}{unroll}\right)$$

$$power_{eff}(f_{div}) = \frac{flop}{\frac{model(f_{div})}{freq} \times power(f_{div})}$$

With $matrix\_size = 160$ elements (each matrix is 160×160 elements big), $body = 43$ cycles, and $unroll = 1$ for the case shown on Figure 3.

For this BLAS-3 kernel, as expected, the computation time is bigger than accessing memory, thus, $model(f_{div}) = cycles_{comp}(f_{div})$)

D. Power efficiency optimization

Our objective in this section is to show that thanks to the performance model we built, the frequency scaling that optimizes power efficiency can be selected. Then the higher performance version is chosen among the most power efficient versions.

We can see that the dot and matrix-vector products are memory bound while the matrix-matrix product is compute bound. Power efficiency is measured through the ratio of GFlops/W. The best frequency optimizing power efficiency of those two kind of code are different. For the case of memory bound codes, the core frequency can be reduced by a large divisor as performance is limited by memory bandwidth which is not very sensitive to core frequency. On the contrary, for computation bound codes, the performance in Gflops decreases linearly with the frequency.

Figures 1c, 2c and 3c represent power efficiency in GFlops/W for respectively dot, matrix-vector and matrix-matrix products. They show that our performance model is similar to the measured performance (from which we deducted power efficiency). Power efficiency for matrix-matrix product is optimal from a frequency divisor of 5, to 16. Among those scalings, the best performance is obtained for the scaling of 5 according to Figure 3a. For the dot product 1c, codes are more energy efficient using a frequency scaling of 5, and their efficiency increases slowly as frequency is reduced. According to our performance model, around 25% of Gflops/W is gained from a frequency divisor of 5 to a frequency divisor of 16, and for this change, the time to execute the kernel has been multiplied by a factor 2.33 (according to our model). In reality, these factors measured are higher than those predicted by the model, but the frequency values for optimal energy efficiency, or some tradeoff between efficiency and performance are the same. Note that for divisor lower than 5, energy efficiency changes more dramatically since the voltage also changes.

We have chosen to show how to optimize energy efficiency, but as our model predicts both running time and power consumption for each frequency, it is easy to build any other metric depending on power and runtime and optimize it. Indeed using this model allows to compute the metric to optimize for each frequency divisor and then to choose the one that fits the best the requirement. Even with a very simple model as we presented, we can predict the running time of simple computational kernels within an error of 38% in the worst case.

Our energy efficiency model is interesting because it shows exactly the same inflection points as the curve of the actual execution. This point allows us to predict what is the best core frequency in order to optimize the power efficiency of the target kernel.

It is also interesting to see that even with a longer running time all the kernels (even matrix multiplication which is compute bound) benefits from frequency reduction. This is caused by the following facts:

- The run time of such kernels is proportional to the frequency;
- The power consumption is also proportional to the fre-

quency.

So the energy efficiency does not depend on the core frequency. But the 3 firsts step of frequency reduction also reduce the voltage which has an huge impact on power consumption.

IV. Related Work

Power efficiency is a hot topic in the HPC community and has been the subject of numerous studies, and the Green500 List is released twice a year. Studies carried out at Carnegie Mellon University in collaboration with Intel [6] have already shown that the SCC is an interesting platform for power efficiency. Philipp Gschwandtner *et al.* also performed an analysis of power efficiency of the Single-chip Cloud computer in [11]. However, this work focuses on benchmarking, while our contribution aims at predicting performance according to a theoritical proposed model.

Performance prediction in the context of frequency and voltage scaling has also been actively investigated [5], [10], [12], and the model usually divides the execution time into memory (or bus, or off-chip) [8], [9], instruction and core instruction, as we did in this paper.

Our contribution is slightly different from usual approach as we do not use any runtime information to predict the impact of frequency and/or voltage scaling on performance. As we use static code analysis to predict performance of a kernel, this could be done at compile time it and does not increase the complexity of runtime system. Static Performance prediction has also been used in the context of autotuning. Yotov *et al.* [13] have shown that performance models, even when using cache hierarchy, could be used to select the version of code with higher performance. Besides, In [3], the authors have shown that a performance model, using measured performance of small kernels, is accurate enough to generate high performance library codes, competing with hand-tune library codes. This demonstrates that performance models can be used in order to compare different versions, at least for regular codes (such as linear algebra codes).

V. Future work

The next step for this study is to extend the performance model presented in this paper to parallel kernels. This is much easier on the SCC Chip than on more classical architectures as the cache access time is constant because of its non-coherence. Bandwidth taken by cache-coherency protocol and possible contention are difficult to model in general. Moreover memory contention on NUMA architecture is a difficult problem. Indeed in such architectures, memory contention not only depends on the memory access pattern but also on the process placement. Philipp Gschwandtner *et al.* showed how memory contention on a single memory controller when several cores are accessing it [11]. We believe it would be very interesting to lead the same experiments for several sets of core frequency. Indeed reducing the core frequency could lead to reducing the stress on the memory controller by spacing memory requests.

Also we would like to improve the model in order to take into account that applications are usually composed of several phases, some compute bound phases followed by others that might be memory bound. Enlarging our model to predict what would be the best frequency for each of those phases.

It would also be interesting to develop a framework, inside a compiler or a performance tuning tool such as MAQAO [2], in order to perform the code analysis automatically. This would reduce the time to build the model for new codes, allowing us to do it on a large number of codes.

VI. Conclusion

We have described a method to predict performance of some linear algebra codes on the Single-chip Cloud Computing architecture. This model can predict performance of a given code for all available frequency divisor and using the known relation between frequency scaling and voltage, it can also predict power efficiency. Based on this prediction we can choose what will be the best frequency to run the kernel. We have shown that we can save energy through this method, but it is actually even more powerful: using the running time prediction and the power model we can choose the frequency in order to optimize either the running time, or the power consumption, or the energy efficiency.

References

[1] The scc programmer's guide, 2011.

[2] Denis Barthou, Andres Charif Rubial, William Jalby, Souad Koliai, and Cdric Valensi. Performance tuning of x86 openmp codes with maqao. In Matthias S. Mller, Michael M. Resch, Alexander Schulz, and Wolfgang E. Nagel, editors, *Tools for High Performance Computing 2009*, pages 95–113. Springer Berlin Heidelberg, 2010.

[3] Denis Barthou, Sebastien Donadio, Alexandre Duchateau, Patrick Carribault, and William Jalby. Loop optimization using adaptive compilation and kernel decomposition. In *ACM/IEEE Intl. Symp. on Code Optimization and Generation*, pages 170–184, San Jose, California, March 2007. IEEE Computer Society.

[4] S. Borkar. The exascale challenge. In *VLSI Design Automation and Test (VLSI-DAT), 2010 International Symposium on*, pages 2 –3, april 2010.

[5] Matthew Curtis-Maury, Filip Blagojevic, Christos D. Antonopoulos, and Dimitrios S. Nikolopoulos. Prediction-based power-performance adaptation of multithreaded scientific codes. *IEEE Trans. Parallel Distrib. Syst.*, 19:1396–1410, October 2008.

[6] R. David, P. Bogdan, R. Marculescu, and U. Ogras. Dynamic power management of voltage-frequency island partitioned networks-on-chip using intel's single-chip cloud computer. In *Networks on Chip (NoCS), 2011 Fifth IEEE/ACM International Symposium on*, pages 257 –258, may 2011.

[7] Agner Fog. Instruction tables lists of instruction latencies, throughputs and micro-operation breakdowns for intel, amd and via cpus. http://www.agner.org/optimize/, 2011.

[8] R. Ge and K.W. Cameron. Power-aware speedup. In *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*, pages 1 –10, march 2007.

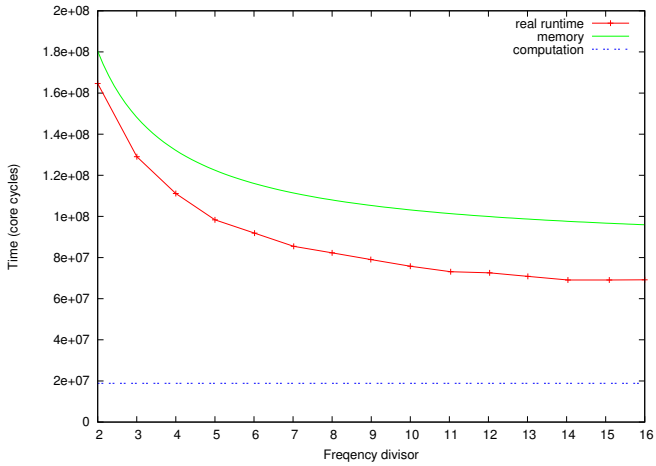
[9] Sang jeong Lee, Hae kag Lee, and Pen chung Yew. Runtime performance projection model for dynamic power management. In *Asia-Pacific Computer Systems Architectures Conference*, pages 186–197, 2007.

[10] Georgios Keramidas, Vasileios Spiliopoulos, and Stefanos Kaxiras. Interval-based models for run-time dvfs orchestration in superscalar processors. In *Conf. Computing Frontiers*, pages 287–296, 2010.

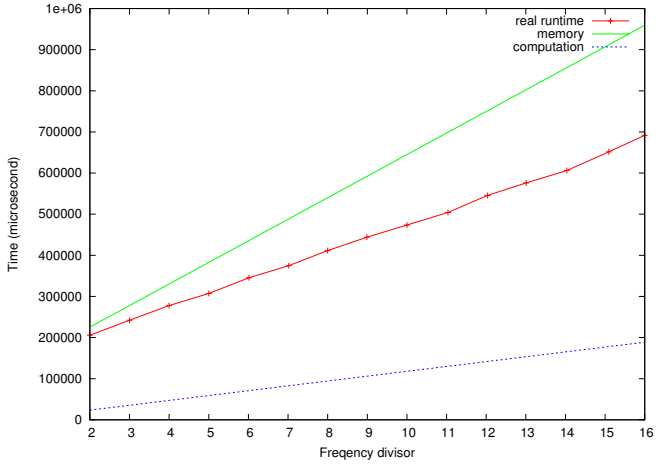
[11] Radu Prodan Philipp Gschwandtner, Thomas Fahringer. Performance analysis and benchmarking of the intel scc. In *Conference on Cluster Computing*, pages 139–149, 2011.

[12] B. Rountree, D.K. Lowenthal, M. Schulz, and B.R. de Supinski. Practical performance prediction under dynamic voltage frequency scaling. In *Green Computing Conference and Workshops (IGCC), 2011 International*, pages 1 –8, july 2011.

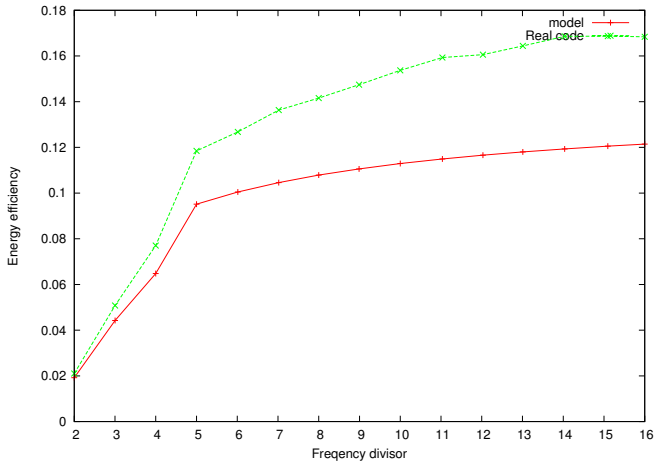
[13] K. Yotov, X. Li, G. Ren, M. Cibulskis, G. DeJong, M. Garzaran, D. Padua, K. Pingali, P. Stodghill, and P. Wu. A comparison of empirical and model-driven optimization. In *ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI'03)*, pages 63–76, San Diego, CA, June 2003.



(a) Dot product: the cycle count is shown according to the core frequency divisor

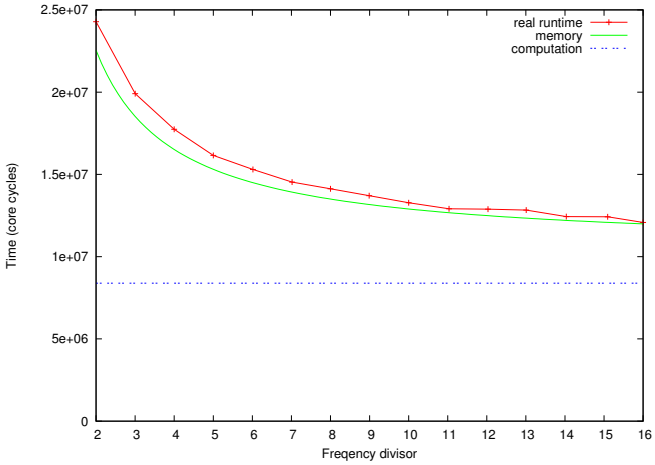


(b) Dot product: runtime in microsecond depending on the core frequency divisor

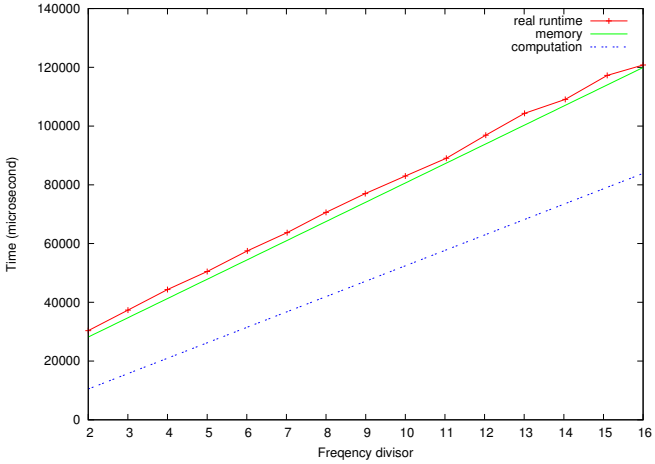


(c) Dot product: power efficiency (in GFlops/W) depending on the core frequency divisor

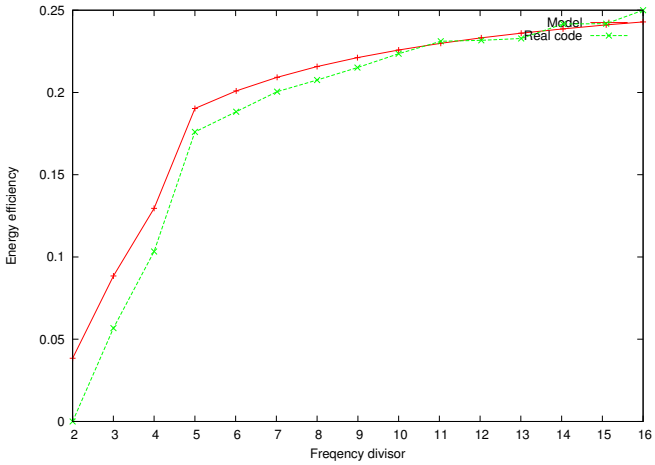
Fig. 1: Vector dot product model: sequential dot product with 2 vectors of 16 MB.



(a) Matrix-vector product: the cycle count is given according to the core frequency divisor.

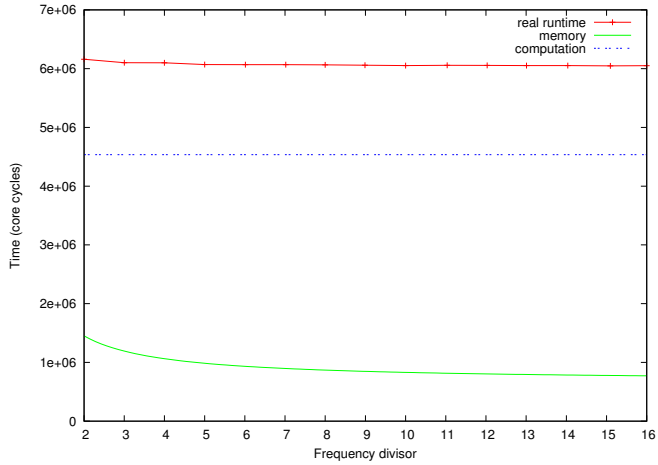


(b) Matrix-vector product: the execution time is given in microsecond depending on the core frequency divisor

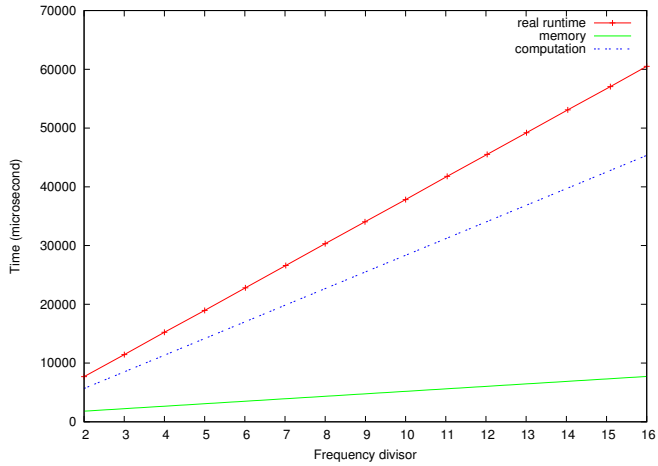


(c) Matrix-vector product: power efficiency (in GFlops/W) depending on the core frequency divisor

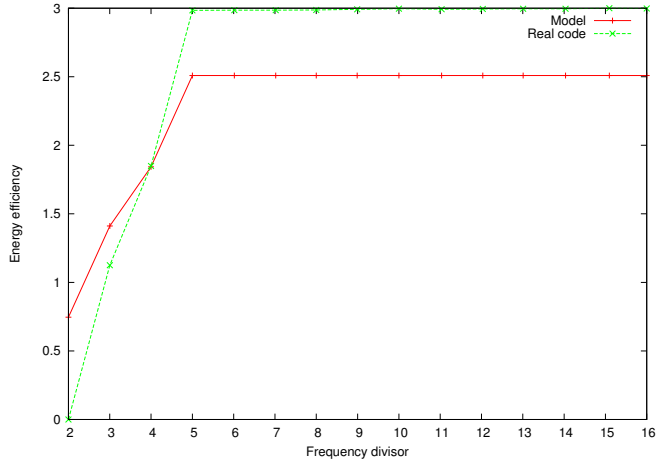
Fig. 2: Matrix-vector multiplication model: sequential code with a 512 by 1024 element size matrix.



(a) Matrix-matrix product model: the cycle count is given according to the the core frequency divisor



(b) Matrix matrix product model: the time in microsecond depending on the core frequency divisor



(c) Matrix-matrix product model: power efficiency (in GFlops/W) depending on the core frequency divisor

Fig. 3: Matrix-matrix multiplication model: sequential code with two matrices of 160 by 160 elements.