Towards a quantum algorithm for evaluating WCETs

Gabriella Bettonte[a], Stéphane Louise[a] and Renaud Sirdey[a]

[a]*Université Paris-Saclay, CEA List*

Abstract

In this paper we propose a quantum-based solution to the problem of counting the cache hits, an important issue when analyzing real-time embedded applications. This field has already seen the development of "quantum-inspired" classical algorithms which are competitive with the state of the art. We designed a polynomial-time dynamic programming algorithm for computing the lowest number of cache hits realized by a deterministic sequence of memory accesses, in the presence of preemptions. Our contribution consists in porting that algorithm to the quantum framework, improving the complexity of the algorithm from $O(N^3)$ to $O(N^2 + N)$.

Keywords

WCETs, preemption, quantum computing, dynamic programming.

1. Introduction

For the most part, the interest addressed to quantum computing comes from the ability of qubits to store a superposition state that reflects all the possible inputs/outputs of a given algorithm, until a measurement is done. This property is called *quantum parallelism* and can, in certain cases, give a massive performance boost for quantum algorithms. But the advantages of quantum computing do not come without caveats: only some classes of problems can be solved by quantum computing, with a definite gain in terms of efficiency with respect to classical computing. Indeed, one important research issue related to quantum computing is defining with precision such problems. Yet, within the variety of quantum algorithms we can identify two main design blueprints. The first one is defined by quantum algorithms capable to reach an exponential speedup over classical algorithms on precisely defined and heavily structured mathematical problems, like for instance Shor's algorithm for integer factorization. The second one is defined by quantum algorithms with a quadratic speedup over classical ones, like for instance Grover's algorithm for searching an unstructured array. Still, a lot of questions remain open about the real world applications and benefits of quantum computing, especially for more arbitrary problems and fields without any a priori "quantum-friendlyness".

In this paper, we consider the problem of worst-case execution time (WCET) evaluation by static analysis of programs. The WCET analysis is highly relevant to the design of safety-critical real-time systems, which needs to respect all the timing constraints they are specified to meet for the verification of their safety properties. Also, WCET analysis has recently seen the developments of "quantum-inspired" classical algorithms which are competitive (in terms of

2nd Quantum Software Engineering and Technology Workshop | IEEE Quantum Week 2021

precision and efficiency) with state-of-the-art approaches. Furthermore, the problems arising in WCET evaluation cover a wide-range of complexity classes, from undecidability in the general case down to NP-hardness and polynomial-time solvability in some restricted cases [17]. As such, it appears to provide a relevant playground to put the quantum computing promise to the test, although other fields may be as relevant and *should* be explored as well.

In this direction, in this paper, we tackle only a restricted setup with the most simple program model: the evaluation of the worst-case number of cache misses of programs performing deterministic sequences of memory accesses, in case of arbitrary preemptions. A preemption is the act of interrupting one task to allow the execution of another task on a machine. There is indeed a strong connection between the number of cache misses done by program and its execution time, as uncached memory accesses are highly time-consuming on modern processors. In other words, we consider a single path program (linear sequences of instructions) and the complexity of the model comes from the arbitrary-placed interruptions, due to other programs running on the same system, which create interferences in the cache memory. For our model, we consider K preemption points and they can occur at any time in the sequence, making the cache behaviour non-predictable.

In this paper, we propose a dynamic programming classical algorithm, with polynomial-time complexity, to compute the minimal number of cache hits in the sequence of memory accesses, as a proxy measure of the WCET for the considered program. Although our model is standard the WCET community, we consider it, as well as the classical polynomial-time algorithm solving it, as the basis to derive a lower complexity (still polynomial-time) hybrid quantum-classical algorithm. In doing so, we demonstrate a first benefit of explicitly using the quantum computing paradigm to the field of WCET calculations, albeit in simplified setting.

Dynamic programming is an algorithmic technique to solve a problem consisting in finding one optimal solution by solving a family of easier sub-problems. We designed a dynamic programming algorithm to compute the minimal number of cache hits while executing a deterministic sequence with K preemptions. This technique to solve WCETs is consistent with others classical solutions in the literature [20]. Still, we want to emphasize that designing the best classical algorithm to solve the considered problem is not the point of our work: we are focused on the portability of classical algorithms to the quantum framework.

This paper is organized as follows: Section II provides a brief overview about cache memories and preemptions. Then, Section III places the paper in the context of the state-of-the-art, and presents our program model. In Section IV we propose a dynamic programming algorithm for evaluating WCET. Section V then contains the quantum version of that algorithm. Lastly, in Section VI we compare the complexity of the two algorithms and in Section VII we provide some perspectives for future works.

2. Background on cache and preemption

Cache memories impact the variability of execution times, since the access time between an element stored in the cache memories and an element which is not, can be up to two orders of magnitude. As cache memories are limited in size, the hardware uses the history of previous accesses in the cache to decide which currently stored elements should be replaced when a new

one must be stored in the cache memory. For instance, LRU (Least Recently Used) cache policy privileges, as replacement candidate, the oldest used line of memory in a set. In the general case, when an access to an element in memory performed by a program is already in the cache, we call it a "cache hit", otherwise we say that a "cache miss" occurred. Cache misses impact on the execution time because the missing element needs to be fetched from the main memory, which induces additional delays.

The advantage of preemptions is the possibility to make optimal utilization of the computing power. In particular, in fully preemptive systems, as the one we present in this paper, the running task can be interrupted at any time by another task with higher priority and be resumed to continue when all higher priority tasks have completed [6]. When the task is preempted, the memory blocks corresponding to the task are usually considered as flushed from the cache memory[1], between the time the task is preempted and the time the task resumes execution. Therefore a substantial amount of time is spent to restore the previous content of the cache, greatly increasing the task's execution time [3]. In certain cases, preemption has not a great impact because it happens at a moment of the program's execution in which data stored into the cache are not useful: for instance, before a cache miss. Still, in general, preemption damages program locality and therefore it causes a degradation of system predictability, making WCETs not easy to characterize and predict [4] [5] [3] [6].

3. Deterministic memory access with preemptions

We consider programs that perform deterministic accesses to the memory, that can be interrupted at any time, using the count of cache misses as a first proxy evaluation of the WCET (or as the literature calls it, the Cache memory preemption delay). A "dual" approach has already been explored in the literature, with papers about applying static analysis techniques to quantum algorithms to evaluate their performance and formally analyze their functional properties [18, 19]. On the other hand, the applications of quantum computing to improve the static analysis of programs has been only scarcely explored [1] and even lesser so is the issue of static analysis of cache misses and WCET where only "quantum-inspired" classic algorithm have been proposed [2]. So a lot of questions remain open about the applications and benefits of quantum computing for software engineering issues [1]. Numerous research works exist in the direction of improving classical polynomial algorithms with a quantum inspired approach, allowing to gain a polynomial factor of complexity [7, 8, 9, 10, 11, 12, 13, 14]. In this paper, we propose a dynamic programming algorithm for computing the minimal number of cache hits and improve it into a quantum-classic hybrid version, leading to a lower complexity.

As model case, we take into consideration the problem of evaluating the WCET of a deterministic sequence of memory accesses, in case of preemptions. We denote the sequence of memory accesses as $a_0, a_1, \dots, a_{N-1}$ and we are supposed to know if each of them is a cache miss or a cache hit in an execution without preemptions (this is done in linear time as a pre-

[1] Whilst it is possible that, for a given set of tasks, preemption would preserve some cache lines, the situation when one wants to calculate WCET of a task in the general case requires the hypothesis that no useful cache line would be retained by a preemption.

processing step). We denote $X[i]$ the i-th access of the sequence and

$$X[i] = \begin{cases} 1 \text{ if cache miss} \\ 0 \text{ if cache hit.} \end{cases}$$

P is the set of possible contents of those memory accesses $p_0, \dots, p_{M-1}$. The number of preemptions interrupting the program is a fixed number K. Preemption can happen at any time, this leads to non-predictability of the model. We suppose also that when a preemption occurs, all the content of the cache is flushed from it.

4. Our classical starting point

We designed a dynamic programming algorithm to compute the minimal number of cache hits while executing a deterministic sequence with K preemptions. The minimal number of cache hits corresponds to the maximal number of cache misses.

First at all, we scan the sequence of memory accesses, as it would be without any point of preemption, and we store the information of being a cache hit or miss into an array X: if the considered access is cache miss $X[i] = 1$, $X[i] = 0$ otherwise. We compute also the $N_i[p_t] > i$, the table of indexes of the first access to the object p_t, after the i-th access, into the sequence of memory access. If there is no other access to p_t, we write ∞.

We suppose to know the solution of the problem for all the sequence $a_{i+1}, \cdots, a_{N-1}$ and the number of preemptions k', where $0 < k' < K$. In other words, it is known:

- $S(i+1, pt, k')$: the smallest number of cache hits for the object p_t and k' preemptions;
- $Y(i+1, k') > i+1$: index of the next preemption in this solution.

Algorithm 1 describes how to determine $S(i, p_t, k')$ and $Y(i, k')$ from the previous data, in case of cache miss (i.e if $X[i] = 1$)

Now, Algorithm 2 describes how to determine $S(i, p_t, k')$ and $Y(i, k')$ from the previous data, in case of cache hit (i.e if $X[i] = 0$).

After performing the algorithm we obtain the smallest number of cache hits for each object p_t, when K preemptions occur, in symbols $S(0, p_t, K)$. $\sum_{p_t} S(0, p_t, K)$ is the solution for the problem of computing the smallest number of cache hits for the sequence of memory accesses.

4.1. Example of application of the classic algorithm

As example, we take into consideration a cache of size 2, with $K = 2$ (i.e two preemptions occur). The set of possible contents of the cache is $P = \{p_0, p_1, p_2\} = \{A, B, C\}$. The sequence of memory accesses is $a_0 a_1 a_2 a_3 a_4 = ABABC$. The first two accesses and the last one are cache misses, while the third and fourth accesses are cache hits. It is simple to see that the worst scenario is when the two points of preemption happen immediately before the two cache hits, because in this case the content of the cache is flushed and A and B has to be restored into the cache when re-called lately. In this scenario the algorithm returns $S_0 = 0$ as minimal number

Algorithm 1: Cache hits counter - cache miss case

Result: The choice between A and B, for a fixed k', is the one that minimize $\sum_{k'} S(i, p_t, k')$

i index in the sequence

$0 < k' < K$ number of preemptions;

case A no preemption

for all p_t in P

  for all k'

    if $p_t = a_i$ **then** $S(i, a_i, k') = 1 + S(i+1, a_i, k')$

    $S(i, p_t, k') = S(i+1, p_t, k')$

    $Y(i, k') = Y(i+1, k')$

    end

case B preemption

for all p_t in P

  for all k'

    if $X[N_i[p_t]] = 0$ and $Y(i+1, k'-1) > N_i[p_t]$

    then

    $B = B1 = S(i, p_t, k') = S(i+1, p_t, k'-1) - 1$

    else

    $B = B2 = S(i, p_t, k') = S(i+1, p_t, k'-1)$

    $Y(i, k') = i$

of cache hits, meaning that all the accesses of the sequence are cache misses. If a preemption happens before a cache miss it does not impact in the performances.

After the algorithm performs the first scan of the sequence of memory accesses we have $X = 11001$. The table N is:

$N =$

i \t	0 (A)	1 (B)	2 (C)
0	2	1	4
1	2	3	4
2	∞	3	4
3	∞	∞	4
4	∞	∞	∞

We give below the details only of the two first iterations, being the most representative. In order to obtain the final result, it suffices to execute again the algorithm, until the sequence of accesses is completed.

Algorithm 2: Cache hits counter - cache hit case

Result: The choice between A and B, for a fixed k', is the one that minimize $\sum_{k'} S(i, p_t, k')$

i index in the sequence

$0 < k' < K$ number of preemptions;

case A no preemption

for all p_t in P

for all k'

if $p_t = a_i$

$A = S(i, a_i, k') = 1 + S(i + 1, a_i, k')$

else

$A = S(i, p_t, k') = S(i + 1, p_t, k')$

$Y(i, k') = Y(i + 1, k')$

case B preemption

for all p_t in P

for all k'

if $(X[N_i[p_t]] = 0$ and $Y(i + 1, k' - 1) > N_i[p_t]$and $p_t \neq a_i)$

$B = B1 = S(i, p_t, k') = S(i + 1, p_t, k' - 1) - 1$

else

$B = B2 = S(i, p_t, k') = S(i + 1, p_t, k' - 1)$

$S(i, a_i, k') = S(i + 1, a_i, k' - 1)$

$Y(i, k') = i$

- a4=C

$S(i = 4, p_t, k') =$

t \k'	0	1	2
0 (A)	0	0	0
1 (B)	0	0	0
2 (C)	0	0	0

$Y(i = 4, k') =$

k'	0	1	2
	∞	4	4

- a3=B

Without preemption:

$A = S(i = 3, pt, k') =$

t \k'	0	1	2
0 (A)	0	0	0
1 (B)	1	1	1
2 (C)	0	0	0

With preemption:

$B = S(i = 3, p_t, k') =$

t \k'	0	1	2
0 (A)	-	0	0
1 (B)	-	1	1
2 (C)	-	0	0

Here, note that "-" is equivalent to 0 for our purpose. Final table:

$S(i = 3, p_t, k') =$

t \k'	0	1	2
0 (A)	0	0	0
1 (B)	1	1	1
2 (C)	0	0	0

$Y(i = 3, k') =$

k'	0	1	2
	∞	3	3

5. Going quantum

In this section we present the “quantum version” of the previous algorithm. We suppose to already know the behaviour of the cache in case of program’s execution without preemptions (i.e the array X) and the table N, both deterministic. While searching for the minimal number of cache miss, in the classic algorithm, we perform three nested loops:

- over the sequence memory accesses (N iterations)
- over all the possible objects that can be into the cache (N iterations)
- over all the k' with $0 < k' < K$ (K iterations)

To compute the solution (i.e. the number of cache hits corresponding to K preemption and the last memory access) the algorithm needs all the number of cache hits corresponding to k'

with $0 < k' < K$. We reformulated the dynamic programming classic algorithm into an hybrid quantum-classical algorithm in which the third loop (the one over the number of preemptions) is suitable to be executed on a quantum computer.

In particular, we will give below the details of the first case (when cache miss happens) of our quantum algorithm, being the second case (when cache hit happens) similar. We build a superposed state represented as $|k', S\rangle$ where k' is the number of considered preemptions and S the corresponding number of cache hits. In other words, for each number of preemptions k', it is known the minimum possible number of cache hits while performing the sequence of memory accesses. Considering, for simplicity of notation, the case of our example, S can be either 0 or 1 so we build the superposed state $\sum_{k'} a_p |k', 0\rangle + b_p |k', 1\rangle$, for each $p \in P$. We recall that P is the set of possible contents of the memory slots. Since each block (a k' iteration) is independent from the others, we can consider them one by one. Therefore we perform a projection on the k', so we represent the qubits as $q_p = a_p |0\rangle + b_p |1\rangle$, with $a_p, b_p \in \{0, 1\}$ and $a_p + b_p = 1$. If the considered memory access is a cache miss (i.e $X[i] = 1$), a preemption may (case B) or may not (case A) occur before accessing it. When a preemption does not occur (case A), we simply apply the identity matrix to the superposed state. When a preemption occurs (case B) the situation is more complex and we need to define two different operators, G and D. Those operators are made of the sub-operators $0_2, I_2, S_2$, applied to the single blocks.

$$0_2 = \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix} I_2 = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} S_2 = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \tag{1}$$

$$D = \begin{bmatrix} 0_2 & I_2 & 0_2 & \cdots & 0_2 \\ 0_2 & 0_2 & I_2 & \cdots & 0_2 \\ \vdots & \vdots & \vdots & \ddots & 0_2 \\ I_2 & 0_2 & 0_2 & 0_2 & 0_2 \end{bmatrix}$$

$$G = \begin{bmatrix} 0_2 & I_2 & 0_2 & \cdots & 0_2 \\ 0_2 & 0_2 & I_2 & \cdots & 0_2 \\ \vdots & \vdots & \vdots & \ddots & 0_2 \\ I_2 & 0_2 & 0_2 & 0_2 & 0_2 \end{bmatrix} \times \begin{bmatrix} S_2 & 0_2 & 0_2 & 0_2 \\ 0_2 & S_2 & 0_3 & 0_2 \\ \vdots & \vdots & \ddots & \vdots \\ 0_2 & 0_2 & 0_2 & S_2 \end{bmatrix}$$

In order to maintain reversibility we need to compute all the more outputs of the algorithm respect to the classical algorithm and keep track of them. Starting from the superposed state, we will apply the identity operator to obtain A, we apply G to obtain $B1$ and D to obtain $B2$

The first oracle O_1 has to choose between $B1$ and $B2$, always block by block:

$$\begin{aligned} O_1 : \quad & |q_1, q_2, \cdots, q_M, q_{M+1}, q_{M+2}, \cdots, q_{2M}, c, y\rangle \longrightarrow \\ & |q_1 f \oplus q_{M+1}(f \oplus 1), q_2 f \oplus q_{M+2}(f \oplus 1), \cdots, \\ & q_M f \oplus q_{2M}(f \oplus 1), q_1(1 \oplus f) \oplus q_{M+1} f, \\ & q_2(1 \oplus f) \oplus q_{M+2} f, \\ & \cdots, q_M(1 \oplus f) \oplus q_{2M} f, c \oplus f, y\rangle \end{aligned}$$

where:

$$q_i = a_p |0\rangle + b_p |1\rangle$$

and c a boolean. We have also:

$$y = c_0 |\infty\rangle + c_1 |1\rangle + \cdots + c_N |N\rangle$$

where $c_0, c_1, \cdots, c_N \in \{0, 1\}$ and $c_0 + c_1, \cdots + c_N = 1$. Finally:

$$f = f = (q_1, \cdots, q_{2M}) = \begin{cases} 1 & \text{if condition is true,} \\ 0 & \text{otherwise} \end{cases}$$

$$\text{with condition true if: } y > N_i(p_t)$$

.

Now, calling B the result of the choice between $B1$ and $B2$, we need to choose between A and B, picking the option that minimizes the number of cache hits. The oracle O_2 is an oracle choosing between A and B.

$$\begin{aligned} O_2 : \quad & |q_1, q_2, \cdots, q_M, q_{M+1}, q_{M+2}, \cdots, q_{2M}, c\rangle \longrightarrow \\ & |q_1 f \oplus q_{M+1}(f \oplus 1), q_2 f \oplus q_{M+2}(f \oplus 1), \cdots, \\ & q_M f \oplus q_{2M}(f \oplus 1), q_1(1 \oplus f) \oplus q_{M+1} f, \\ & q_2(1 \oplus f) \oplus q_{M+2} f, \cdots, q_M(1 \oplus f) \oplus q_{2M} f, c \oplus f\rangle \end{aligned}$$

where

$$q_i = a_p |0\rangle + b_p |1\rangle$$

and c is a boolean. Then, we have:

$$f = f = (q_1, \cdots, q_{2M}) = \begin{cases} 1 & \text{if condition is true,} \\ 0 & \text{otherwise} \end{cases}$$

$$\text{with condition true if: } q_1 + \cdots + q_M < q_{M+1} + \cdots + q_{2M}$$

where "+" is the usual sum over qubits, executable by a Quantum Adder [15], [16].

In Fig. 1 it is represented the quantum algorithm's flow.

5.1. Example of application of the quantum algorithm

In this section we give a glimpse of the functioning of our algorithm though a simple example. We suppose to have a sequence of memory accesses of length $N = 3$: ABB. We have $P = \{A, B\}$, then $M = 2$. The choice of this simplified example comes from the fact that the number of qubits needed to perform the quantum algorithm increases quickly with the number of memory accesses. That being said, with such a low N and consequently low K, it is not possible to appreciate the advantage of the quantum algorithm in terms of complexity. We suppose that the cache has one line. This means that in the best scenario the number of cache hits is 1, while in the worst scenario, if a preemption occurs before the third access, the number of cache hits is 0. Finally, we suppose $K = 1$. We can build the deterministic tables N and Y:

$N =$

i \t	0 (A)	1 (B)
0	2	1
1	2	∞
2	∞	∞

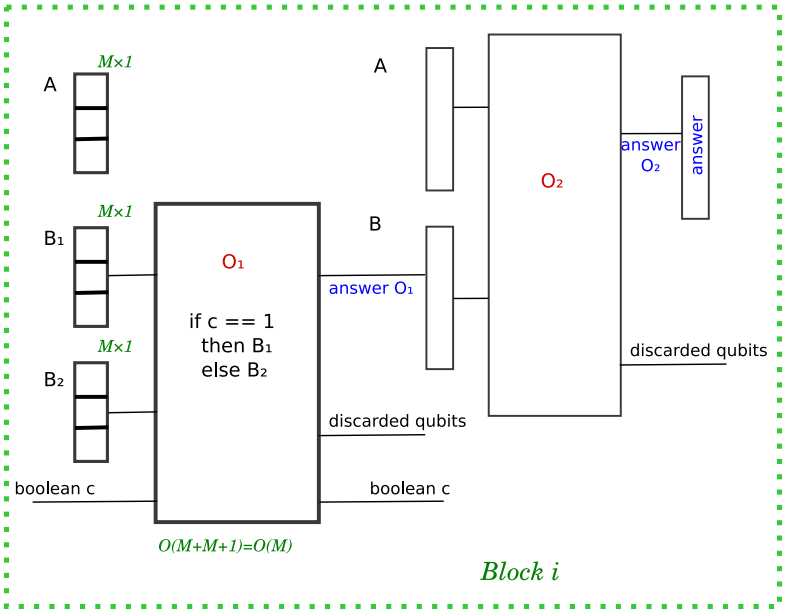


Figure 1: Quantum circuit for a block (i.e. for a fixed k'). Oracle O_1 chooses between $B1$ and $B2$, while O_2 chooses between A and B, where B is the answer of O_1.

We build the superposition representing all possible number of cache hits and all possible number of preemption $|k', S\rangle$ with $k' = \{0, 1\}$ and $S = \{0, 1\}$ for one object in P:

$$\alpha_0 |0, 0\rangle + \alpha_1 |0, 1\rangle + \alpha_2 |1, 0\rangle + \alpha_3 |1, 1\rangle .$$

In this superposition we have that either $\alpha_i = 0$ either $\alpha_{i+1} = 0$, with i even. Then,

$$D = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix}$$

and

$$G = \begin{bmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix}$$

.

In order to build a quantum circuit, we can express the matrices D and G as boolean expression on the qubits a, b:

$$D : |a, b\rangle \longrightarrow |NOTa, b\rangle$$

and

$$G : |a, b\rangle \longrightarrow |NOTa, NOTb\rangle$$

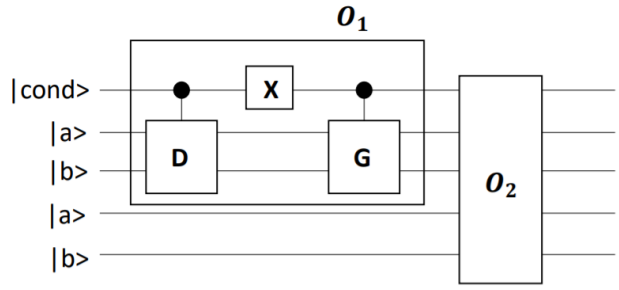


Figure 2: Choice between $B1$ and $B2$

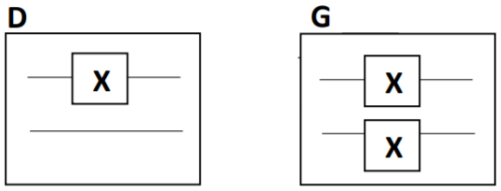


Figure 3: D and G operators

Now, the oracle O_2 has to choose between the case A and the case B, where B is the result of the first oracle O_1.

$$\text{case A } \alpha_0 |0,0\rangle + \alpha_1 |0,1\rangle + \alpha_2 |1,0\rangle + \alpha_3 |1,1\rangle$$

$$\text{case B } \alpha'_0 |0,0\rangle + \alpha'_1 |0,1\rangle + \alpha'_2 |1,0\rangle + \alpha'_3 |1,1\rangle$$

The answer of the O_2 oracle is A if $\alpha_1 + \alpha_3 < \alpha'_1 + \alpha'_3$, B otherwise.

5.2. Post-processing

After applying the previous quantum-algorithm, for each block (i.e. for each k'), we have a superposed state $\sum_{k'} |k', S_0\rangle$ where S_0 is the minimal number of cache hits if k' preemptions occur and $0 < k' < K$. From this superposed state we need to extrapolate the S_0 corresponding to K preemption. The idea is to use a Grover-style searching algorithm over the superposed state to isolate the solution (i.e S_0 for $k' = K$). In particular, we expect to have as input for the oracle the superposition:

$$\begin{aligned}
&0\,|0,0\rangle + 0\,|0,1\rangle + \cdots + \alpha_0\,|0,S_0\rangle + \cdots \\
&+0\,|k',N-1\rangle + 0\,|k',0\rangle + 0\,|k',1\rangle + \cdots \\
&+\alpha_{k'}\,|k',S_0\rangle + \cdots + 0\,|k',N-1\rangle + \cdots \\
&+0\,|K,0\rangle + 0\,|K,1\rangle + \cdots + \\
&+\alpha_K\,|K,S_0\rangle + \cdots + 0\,|K,N-1\rangle
\end{aligned}$$

We suppose then to build an oracle O_G that selects the qubits with "index" corresponding to K. We suppose that O_G implements, in a Grover-style, an Amplitude Amplifier that amplifies coefficients α_i and leaves to zero the zeros coefficients, see Fig.4. If at the end of the algorithm the result is $|x, y\rangle$ where $x \neq K$, it means that the searching algorithm didn't worked and we should apply again the algorithm.

In Fig. 5 we can see that the oracle takes as input superposed state of the results of the circuit for each k' in Fig. 1 and returns the minimal number of cache hits in the sequence of memory access in case of K preemptions.

6. Complexity analysis

In order to compute the final solution, at each iteration of the first loop, we create a table of dimensions $M \times K$, containing the number of the cache hits, for each item in P and each k', with $0 < k' \leq K$. Each iteration of the second loop is independent from the others, meaning that we can compute any row of the next table independently from the others rows. Then, the complexity of "updating" a row (most inner loop) is $O(K)$. Therefore the dynamic programming classical algorithm has a complexity of $O(M \times N \times K)$ (see Fig 6a). In the context of porting classical applications to quantum computing, we designed a "hybrid quantum-classical version" of the previous algorithm, whose output is the number of cache hits corresponding to K preemption. We consider a row in one table as a superposition of states representing all the numbers of preemptions k' and all the possible values of the number of cache hits. In this way, the complexity of the act of "updating" a row becomes $O(1)$ (see Fig. 6b). Therefore the complexity of the resulting quantum algorithm becomes $O(M \times N)$. As a drawback, we need to post-process the result we obtained at the previous step: superpositions of dimension $K \times U$, where $U < N$ is the maximum number of cache hits. However, we only need the value of the number of cache hits when K preemptions occur (i.e. only one value per row in the last set of rows). Therefore, for each row, we can use Grover's algorithm to extract the value of interest in the associated superposition. This means that this post-processing has complexity $\sqrt{K \times U} < \sqrt{K \times N} < \sqrt{N^2} = N$. Hence, the complexity of the whole quantum algorithm is decreased to $O(N^2 + N)$ compared to $O(N^3)$ for the classical algorithm.

7. Conclusions and perspectives

In this paper, as a first step to deal with WCET-related problems by means of quantum-classic hybrid algorithms, we worked on a highly simplified program model since we just considered single-path programs, yet in the presence of arbitrary preemptions. A natural perspective would be to attempt to generalize this approach to more complex program models allowing for some non-determinism in the control-flow. However, such a generalisation would require a careful control of the size of the quantum state representing the cache hit counters array in order to avoid a complexity blow up at the post-processing amplification step.

Also, in this paper, we have ported a polynomial-time dynamic programming algorithm to the quantum framework. In essence, most such algorithms follow the same regular pattern of several nested loops updating an array data structure with the final result obtained in only one

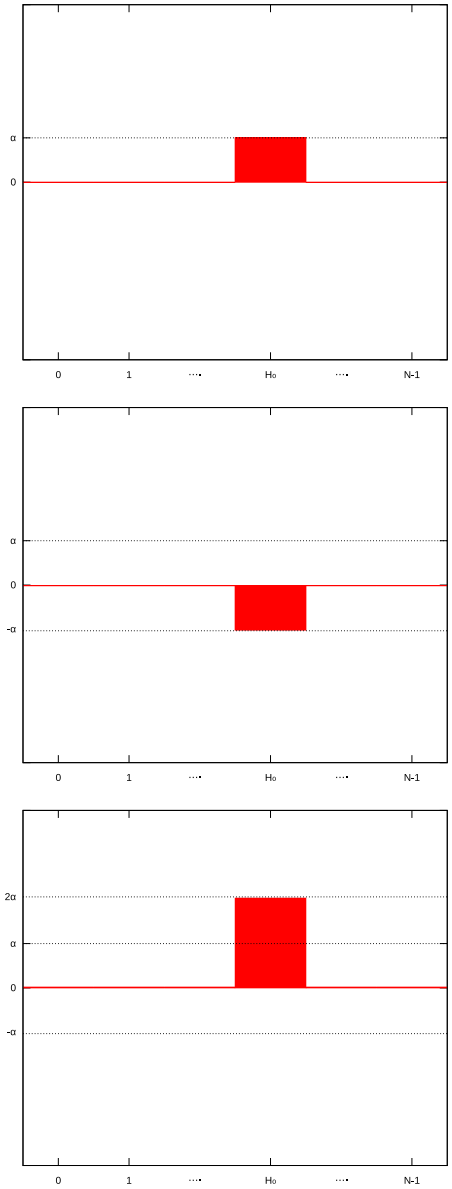


Figure 4: Amplitude amplification performed by the oracle O_G. The coefficient α, the one corresponding to the solution $|K, S_0\rangle$, is amplified while the coefficients 0s, corresponding to the non-solutions, are not modified by the amplification. In the figure we can see how the α is at first reversed and then doubled, leading to an amplification, in a Grover-style fashion.

of the entries of the last array. As such, the approach in this paper, consists in turning the inner loop of our algorithm in a "quantum parallel for" associated to a Grover-style amplification on the *single* result of interest. As a consequence, the approach in this paper may potentially be generalized to other (exponential-time or polynomial-time) dynamic programming algorithms

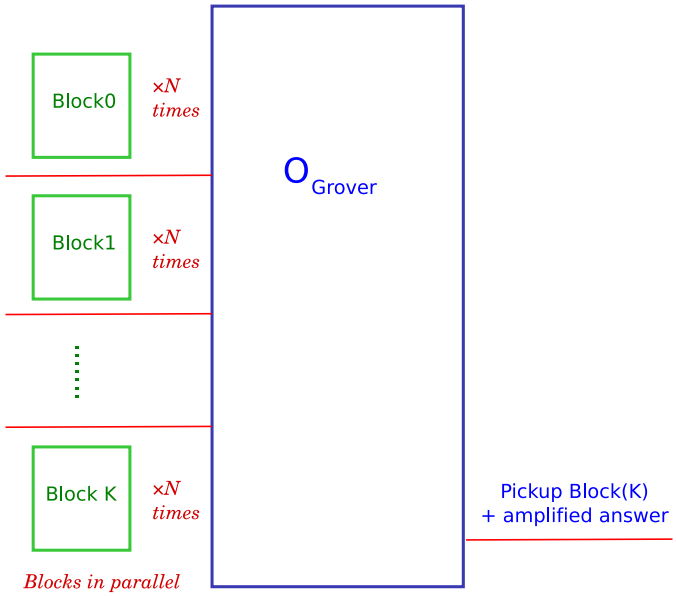


Figure 5: Post-processing circuit. The oracle O_G takes as input the answer of O_2 for each block, after N application of the circuit in Fig.1. It returns as final answer the number of cache hits S_O for the considered memory accesses sequence, in case of K preemptions.

in order to derive (polynomial-only) quantum speedups.

References

[1] R. Hall,"A Quantum Algorithm for Software Engineering Search." 2009.

[2] S. Louise, "A First Step Toward Using Quantum Computing for Low-Level WCETs Estimations." 2019.

[3] C.-G.Lee, J.Hahn, Y.-M.Seo, S.L.Min, R.Ha,S.Hong, C.Y.Park, M. Lee, and C. S. Kim, "Analysis ofcache-related preemption delay infixed-priority preemptive scheduling." 1998.

[4] H. Ramaprasad, F. Mueller, "Tightening the bounds on feasible pre-emption points." 2006.

[5] H. Ramaprasad, F. Mueller, "Bounding worst-case response timefor tasks with non-preemptive regions." 2008.

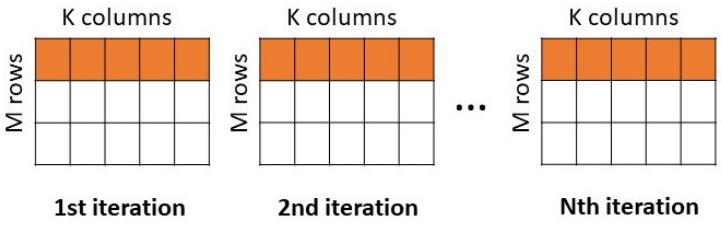
[6] G. Buttazzo, M. Bertogna, G. Yao, "Limited Preemptive Scheduling for Real-Time Systems. A Survey." 2013

[7] A. Manju, M.J. Nigam, "Applications of quantum inspired computational intelligence: a survey".

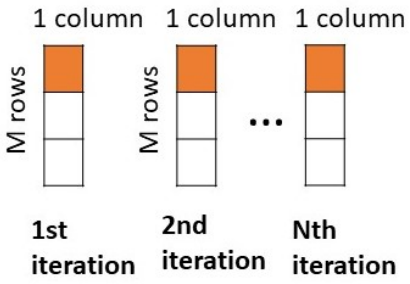
[8] S. B. Ramezani, A. Sommers, H. K. Manchukonda, S. Rahimi and A. Amirlatifi, "Machine Learning Algorithms in Quantum Computing: A Survey," 2020.

[9] E. Tang, "A quantum-inspired classical algorithm for recommendation systems," 2019.

[10] H. Talbi, A. Draa and M. Batouche, "A new quantum-inspired genetic algorithm for solving the travelling salesman problem," 2004.



(a) Classic algorithm



(b) Quantum algorithm

Figure 6: In orange: the number of cache hits, for one object in P and all k'

[11] D. Jethwani and F. Le Gall and Sanjay K. Singh, "Quantum-Inspired Classical Algorithms for Singular Value Transformation," 2020.
[12] G. Ripoll, J. Jose, "Quantum-inspired algorithms for multivariate analysis: from interpolation to partial differential equations," 2021.
[13] N. Chia and H. Lin and C. Wang. , "Quantum-inspired algorithms for multivariate analysis: from interpolation to partial differential equations," 2021.
[14] A. Gilyen and Z. Song and E. Tang, "An improved quantum-inspired algorithm for linear regression," 2020.
[15] G. Florio, D. Picca, "Quantum implementation of elementary arithmetic operations," 2004.
[16] A.V. Cherkas and S.A. Chivilikhin, "Quantum adder of classical numbers," 2016.
[17] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. F.d Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and . Stenström. " The worst-case execution-time problem—overview of methods and survey of tools." 2008
[18] A. J. Abhari, S. Patil, D. Kudrow, J. Heckey, A. Lvov, F. T. Chong et M. Martonosi, "Scaffcc : a framework for compilation and analysis of quantum computing programs." 2014
[19] A. Facon, S. Guilley, M. Lec'Hvien, A. Schaub et Y. Souissi, "Detecting cache-timing vulnerabilities in post-quantum cryptography algorithms." 2018
[20] J. Cavicchio, C. Tessler and N. Fisher, "Minimizing cache overhead via loaded cache blocks and preemption placement." 2015