Computing (2017) 99:865–888
DOI 10.1007/s00607-016-0535-4

A methodology pruning the search space of six compiler transformations by addressing them together as one problem and by exploiting the hardware architecture details

Vasilios Kelefouras[1]

Received: 2 May 2016 / Accepted: 22 December 2016 / Published online: 9 January 2017

Abstract Today's compilers have a plethora of optimizations-transformations to choose from, and the correct choice, order as well parameters of transformations have a significant/large impact on performance; choosing the correct order and parameters of optimizations has been a long standing problem in compilation research, which until now remains unsolved; the separate sub-problems optimization gives a different schedule/binary for each sub-problem and these schedules cannot coexist, as by refining one degrades the other. Researchers try to solve this problem by using iterative compilation techniques but the search space is so big that it cannot be searched even by using modern supercomputers. Moreover, compiler transformations do not take into account the hardware architecture details and data reuse in an efficient way. In this paper, a new iterative compilation methodology is presented which reduces the search space of six compiler transformations by addressing the above problems; the search space is reduced by many orders of magnitude and thus an efficient solution is now capable to be found. The transformations are the following: loop tiling (including the number of the levels of tiling), loop unroll, register allocation, scalar replacement, loop interchange and data array layouts. The search space is reduced (a) by addressing the aforementioned transformations together as one problem and not separately, (b) by taking into account the custom hardware architecture details (e.g., cache size and associativity) and algorithm characteristics (e.g., data reuse). The proposed methodology has been evaluated over iterative compilation and gcc/icc compilers, on both embedded and general purpose processors; it achieves significant performance gains at many orders of magnitude lower compilation time.

✉ Vasilios Kelefouras
kelefouras@ece.upatras.gr
http://www.kelefouras.gr

[1] University of Patras, Patras, Greece

Keywords Loop unroll · Loop tiling · Scalar replacement · Register allocation · Data reuse · Cache · Loop transformations · Iterative compilation

Mathematics Subject Classification 68N20 Compilers and interpreters

1 Introduction

Choosing the correct order and parameters of optimizations has long been known to be an open problem in compilation research for decades. Compiler writers typically use a combination of experience and insight to construct the sequence of optimizations found in compilers. The optimum sequence of optimization phases for a specific code, normally is not efficient for another. This is because the back end compiler phases (e.g., loop tiling, register allocation) and the scheduling sub-problems depend on each other; these dependencies require that all phases should be optimized together as one problem and not separately.

Towards the above problem, many iterative compilation techniques have been proposed; iterative compilation outperforms the most aggressive compilation settings of commercial compilers. In iterative compilation, a number of different versions of the program is generated-executed by applying a set of compiler transformations, at all different combinations/sequences. Researchers and current compilers apply (1) iterative compilation techniques [1–4], (2) both iterative compilation and machine learning compilation techniques (to decrease search space and thus compilation time) [5–10], (3) both iterative compilation and genetic algorithms (decrease the search space) [4,5,11–15], (4) compiler transformations by using heuristics and empirical methods [16], (5) both iterative compilation and statistical techniques [17], (6) exhaustive search [5]. These approaches require very large compilation times which limit their practical use. This has led compiler researchers use exploration prediction models focusing on beneficial areas of optimization search space [10,18–20].

The problem is that iterative compilation requires extremely long compilation times, even by using machine learning compilation techniques or genetic algorithms to decrease the search space; thus, iterative compilation cannot include all existing transformations and their parameters, e.g., unroll factor values and tile sizes, because in this case compilation will last for many many years. As a consequence, a very large number of solutions is not tested.

In contrast to all the above, the proposed methodology uses a different iterative compilation approach. Instead of applying exploration and prediction methods, it fully exploits the hardware (HW) architecture details, e.g., cache size and associativity, and the custom software (SW) characteristics, e.g., subscript equations (constraint propagation to the HW and SW parameters); in this way, the search space is decreased theoretically by many orders of magnitude and thus an efficient schedule is now capable to be found, e.g., given the cache architecture details, the number of different tile sizes tested is decreased. In Sect. 4, I show that if the transformations addressed in this paper (including almost all different transformation parameters) are included to iterative compilation, the compilation time lasts from 10^9 up to 10^{21} years (for the given input sizes). On the other hand, the compilation time of the proposed methodology lasts from some seconds up to some hours. Thus, an efficient schedule can be found fast.

The major contributions of this paper are the following. First, loop unroll, register allocation and scalar replacement are addressed together as one problem and not separately, by taking into account data reuse and RF size (Sect. 3.1). Second, loop tiling and data array layouts are addressed together as one problem, by taking into account cache size and associativity and data reuse (Sect. 3.2). Third, according to the two major contributions given above, the search space is reduced by many orders of magnitude and thus an efficient solution is now capable to be found.

The experimental results have taken by using PowerPC-440 and Intel Xeon Quad Core E3-1240 v3 embedded and general purpose processors, respectively. The proposed methodology has been evaluated for seven well-known data intensive algorithms over iterative compilation and gcc/Intel icc compilers (speedup values from 1.4 up to 3.3); the evaluation refers to both compilation time and performance.

The remainder of this paper is organized as follows. In Sect. 2, the related work is given. The proposed methodology is given in Sect. 3 while experimental results are given in Sect. 4. Finally, Sect. 5 is dedicated to conclusions.

2 Related work

Normally, iterative compilation methods include transformations with low compilation time such as common subexpression elimination, unreachable code elimination, branch chaining and not compile time expensive transformations such as loop tiling and loop unroll. Iterative compilation techniques either do not use loop tiling and loop unroll transformations at all, or they use them only for specific tile sizes, levels of tiling and unroll factor values [21–23]. In [21], one level of tiling is used with tile sizes from 1 up to 100 and unroll factor values from 1 up to 20 (innermost iterator only). In [22], multiple levels of tiling are applied but with fixed tile sizes. In [24], all tile sizes are considered but each loop is optimized in isolation; loop unroll is applied in isolation also. In [23], loop tiling is applied with fixed tile sizes. In [25,26], only loop unroll transformation is applied.

Regarding genetic algorithms, [4,5,11–15] show that selecting a better sequence of optimizations significantly improves execution time. In [12] a genetic algorithm is used to find optimization sequences reducing code size. In [11], a large experimental study of the search space of the compilation sequences is made. They examine the structure of the search space, in particular the distribution of local minima relative to the global minima and devise new search based algorithms that outperform generic search techniques. In [27] they use machine learning on a training phase to predict good polyhedral optimizations.

It is important to say that genetic algorithms are not able to solve the phase ordering problem but only to predict an efficient sequence of optimizations. Moreover, they require a very large compilation time and they are difficult to implement.

Kulkarni et al. [5] first reduces the search space by avoiding unnecessary executions and then modifies the search space so fewer generations are required. In [28] they suppose that some compiler phases may not interact with each other and thus, they are removed from the phase order search space (they apply them implicitly after every relevant phase).

Kulkarni and Cavazos [29] uses an artificial neural network to predict the best transformation (from a given set) that should be applied based on the characteristics of the code. Once the best transformation has been found, the procedure is repeated to find the second best transformation etc. In [6] prediction models are given to predict the set of optimizations that should be turned on. [18] address the problem of predicting good compiler optimizations by using performance counters to automatically generate compiler heuristics.

An innovative approach to iterative compilation was proposed by [30] where they used performance counters to propose new optimization sequences. The proposed sequences were evaluated and the performance counters are measured to choose the new optimizations to try. In [27], they formulate the selection of the best transformation sequence as a learning problem, and they use off-line training to build predictors that compute the best sequence of polyhedral optimizations to apply.

In contrast to all the above works, the proposed methodology methodology uses a different approach. Instead of applying exploration and prediction methods, the search space is decreased by fully utilizing the HW architecture details and the SW characteristics.

As far as register the allocation problem is concerned, many methodologies exist such as [31–37]. In [31–35], data reuse is not taken into account. In [36,37], data reuse is taken into account either by greedily assigning the available registers to the data array references or by applying loop unroll transformation to expose reuse and opportunities for maximizing parallelism. In [38], a survey on combinatorial register allocation and instruction scheduling is given. Finally, regarding data cache miss elimination methods, much research has been done in [39–45].

3 Proposed methodology

The proposed methodology takes the target C-code and HW architecture details as input and automatically generates only the efficient schedules, while the inefficient ones are being discarded, decreasing the search space by many orders of magnitude, e.g., all the schedules using a larger number of registers than the available are discarded. Then, searching only among a specific set of efficient schedules is applied and an efficient schedule can be found fast.

The initial search space is shown in Fig. 1; for a two level cache architecture it includes one level of tiling (tiling for the L1 or L2 cache), two levels of tiling (tiling for both L1 and L2 cache) and no tiling, schedules/binaries; loop tiling is applied to all the iterators. Also it includes register allocation, scalar replacement, loop unroll to all the iterators, loop interchange, and different data array layouts. In Fig. 1, there are seven different problems/transformations and thus the search space consists up to $7! = 5040$ transformation combinations. In Sect. 4, I show that if the transformations presented in Fig. 1 (including almost all different transformation parameters) are included to iterative compilation, the search space is from 10^{17} up to 10^{29} schedules(for the given input sizes); given that $1s = 3.17 \times 10^{-8}$ years and supposing that compilation time takes 1 sec, the compilation time is from 10^{9} up to 10^{21} years. On the other hand, the

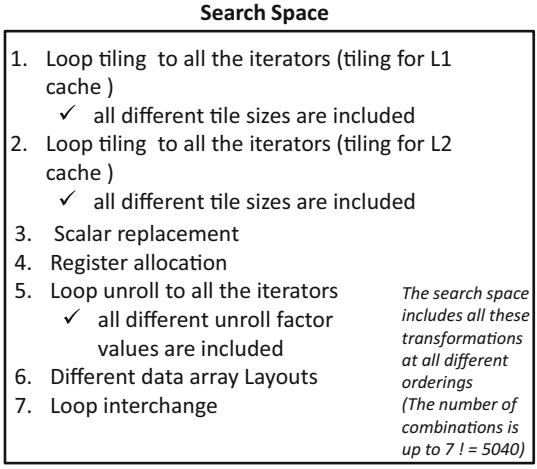

Fig. 1 Search space being addressed

proposed methodology decreases the search space from 10^1 up to 10^5 schedules. In this way, the search space is now capable to be searched in a short amount of time.

Regarding target applications, this methodology optimizes loop kernels; as it is well known, 90% of the execution time of a computer program is spent executing 10% of the code (also known as the 90/10 law) [46]. The methodology is applied to both perfectly and imperfectly nested loops, where all the array subscripts are linear equations of the iterators (which in most cases do); an array subscript is another way to describe an array index (multidimensional arrays use one subscript for each dimension). This methodology can also be applied to C code containing SSE/AVX instructions (SIMD). For the reminder of this paper, I refer to architectures having separate L1 data and instruction cache (vast majority of architectures). In this case, the program code always fits in L1 instruction cache since I refer to loop kernels only, whose code size is small; thus, upper level unified/shared caches, if exist, contain only data. On the other hand, if a unified L1 cache exists, memory management becomes very complicated.

The proposed methodology is shown in Fig. 2. First, parsing is applied in order to extract the custom software characteristics (loop kernel parameters), i.e., data dependences, array references, subscript equations, loop iterators and bounds and iterator nesting level values. These characteristics are used to apply the aforementioned transformations in an efficient way. One mathematical equation is created for each array's subscript in order to find the corresponding memory access pattern, e.g., $(A[2*i+j])$ and $(B[i, j])$ give $(2*i+j=c1)$ and $(i=c21$ and $j=c22)$, respectively, where $(c1, c21, c22)$ are constant numbers and their range is computed according to the iterator bound values. Each equation defines the memory access pattern of the specific array reference; data reuse is found by these equations (data reuse occurs when a specific memory location is accessed more than once).

Regarding 2-d arrays, two equations are created and not one because the data array layout is not fixed, e.g., regarding $B[i, j]$ reference, if $(N*i+j=c)$ (where N is the

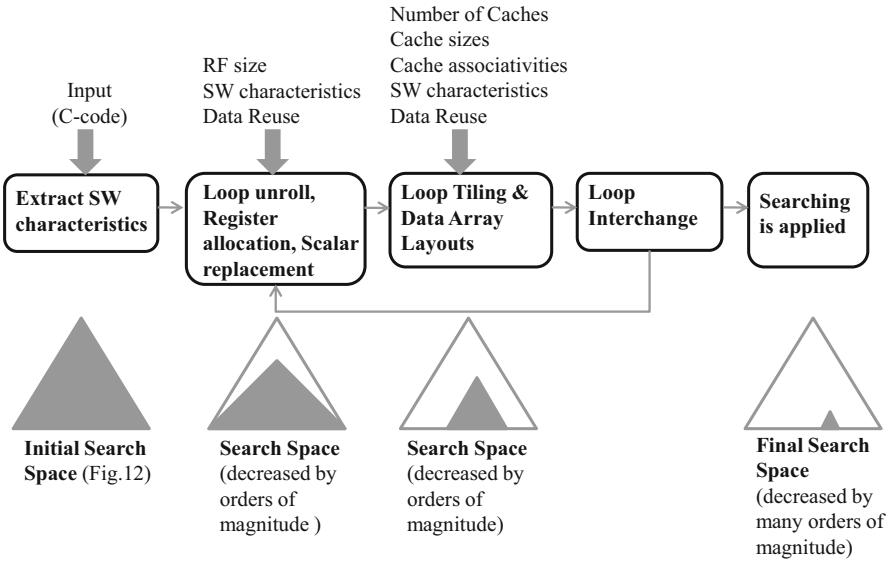

Fig. 2 Flow graph of the proposed methodology

number of the array columns) is taken instead of ($i = c1$) and ($j = c2$), then row-wise layout is taken which may not be efficient.

Definition 1 Subscript equations which have more than one solution for at least one constant value, are named type2 equations. All others, are named type1 equations, e.g., ($2 * i + j = c1$) is a type2 equation, while ($i = c21$ and $j = c22$) is a type1 equation.

Arrays with type2 subscript equations are accessed more than once from memory (data reuse), e.g., ($2i + j = 7$) holds for several iteration vectors; on the other hand, equations of type1 fetch their elements only once; in [47], I give a new compiler transformation that fully exploits data reuse of type2 equations. However, both type1 and type2 arrays may be accessed more than once in the case that the loop kernel contains at least one iterator that does not exist in the subscript equation, e.g., consider a loop kernel containing k, i, j iterators and $B[i, j]$ reference; $B[i, j]$ is accessed as many times as k iterator indicates.

Definition 2 The subscript equations which are not given by a compile time known expression (e.g., they depend on the input data), are further classified into type3 equations. Data reuse of type3 arrays cannot be exploited, as the arrays elements are not accessed according to a mathematical formula.

After the SW characteristics have been extracted, I apply loop unroll, register allocation and scalar replacement, in a novel way, by taking into account the subscript equations, data reuse and RF size (Sect. 3.1). I generate extra mathematical inequalities that give all the efficient RF tile sizes and shapes, according to the number of the available registers. All the schedules using a different number of registers than

those the proposed inequalities give are not considered, decreasing the search space. Moreover, one level of tiling is applied for each cache memory (if needed), in a novel way, by taking into account the cache size and associativity, the data array layouts and the subscript equations (Sect. 3.2). One inequality is created for each cache memory; these inequalities contain all the efficient tile sizes and shapes, according to the cache architecture details. All tile sizes and shapes and array layouts different than those the proposed inequalities give are not considered decreasing the search space.

It is important to say that partitioning the arrays into tiles according to the cache size only is not enough because tiles may conflict with each other due to the cache modulo effect. Given that all the tiles must remain in cache, (a) all the tile elements have to be written in consecutive main memory locations and therefore in consecutive cache locations (otherwise, the data array layout is changed), and (b) different array tiles have to be loaded in different cache ways in order not to conflict with each other. All the schedules with different tile sizes than those the proposed inequalities give are not considered, decreasing the exploration space even more.

In contrast to iterative compilation methods, the number of levels of tiling is not one, but it depends on the number of the memories. For a two level cache architecture, loop tiling is applied for (a) L1, (b) L2, (c) L1 and L2, and (d) none. The number of levels of tiling is found by testing as the data reuse advantage can be overlapped by the extra inserted instructions; although the number of data accesses is decreased (by applying loop tiling), the number of addressing instructions (and in several cases the number of load/store instructions) is increased.

When the above procedure has ended, all the efficient transformation parameters have been found (according to the Sects. 3.1 and 3.2) and all the inefficient parameters have been discarded reducing the search space by many orders of magnitude. Then, all the remaining schedules are automatically transformed into C-code; the C-codes are generated by applying the aforementioned transformations with the parameters found in Sects. 3.1 and 3.2. Afterwards, all the C-codes are compiled by the target architecture compiler and all the binaries run to the target platform to find the fastest.

The remainder of the proposed methodology has been divided into two sections describing in more detail the most complex steps of Fig. 2.

3.1 Loop unroll, scalar replacement, register allocation

Loop unroll, register allocation, scalar replacement, data reuse and register file (RF) size, strongly depend on each other and thus they are addressed together, as one problem and not separately. The reason follows. Loop unroll is applied in order to expose common array references in the loop body which they are then replaced by variables/registers in order to decrease the number of L/S and addressing instructions; exposing common array references in the loop body exposes data reuse (In Fig. 3b, loop unroll is applied to the first two iterators with unroll factor values equal to 2, while in Fig. 3c the common array references are replaced by variables). However, the number of common array references exposed depends on the iterators that loop unroll is being applied to and on the loop unroll factor values. Unrolling the correct iterator exposes data reuse; on the other hand, in Rule 2, I show that data reuse cannot be

(a)

```
for (row = 2; row < N-2; row++) {
 for (col = 2; col < M-2; col++) {
   tmp=0;
   for (mr=0; mr<5; mr++) {
     for (mc=0; mc<5; mc++) {
tmp+=in[row+mr-2][col+mc-2]*mask[mr][mc] ;
               } }
out[row][col]=tmp/159;
              }}
```

(b)

```
for (row = 2; row < N-2; row+=2) {
 for (col = 2; col < M-2; col+=2) {
tmp1=0; tmp2=0; tmp3=0; tmp4=0;
  for (mr=0; mr<5; mr++) {
    for (mc=0; mc<5; mc++) {
tmp1 += in[row+mr-2][col+mc-2] * mask[mr][mc];
tmp2+= in[row+mr-2][col+mc-1] * mask[mr][mc];
tmp3 += in[row+mr-1][col+mc-2] * mask[mr][mc];
tmp4 += in[row+mr-1][col+mc-1] * mask[mr][mc];
                        } }
out[row][col]=tmp1/159;
out[row][col+1]=tmp2/159;
out[row+1][col]=tmp3/159;
out[row+1][col+1]=tmp4/159;
}}
```

(c)

```
for (row = 2; row < N-2; row+=2) {
 for (col = 2; col < M-2; col+=2) {
  out0=0;out1=0;out2=0;out3=0;
     for (mr=0; mr<5; mr++) {
       addr1=row+mr-2;
     for (mc=0; mc<5; mc++) {
       addr2=col+mc-2;
       reg=mask[mr][mc];
in1=in[addr1][addr2];
out0+=(in1*reg);
in1=in[addr1][addr2+1];
out1+=(in1*reg);
in1=in[addr1+1][addr2];
out2+=(in1*reg);
in1=in[addr1+1][addr2+1];
out3+=(in1*reg);
} }
out[row][col]=out0/159;
out[row][col+1]=out1/159;
out[row+1][col]=out2/159;
out[row+1][col+1]=out3/159;
}}
```

(d)

```
for (row = 2; row < N-2; row++) {
 for (col = 2; col < M-2; col+=6) {
out0=0;out1=0;out2=0;out3=0;
out4=0;out5=0;
   for (mr=0; mr<5; mr++) {
            addr1=row+mr-2;
            in0=in[addr1][col-2];
            in1=in[addr1][col-1];
            in2=in[addr1][col];
            in3=in[addr1][col+1];
            in4=in[addr1][col+2];
     for (mc=0; mc<5; mc++) {
        reg_mask=mask[mr][mc];
         in5=in[addr1][col+3+mc];
out0 += (in0*reg_mask);
out1 += (in1*reg_mask);
out2 += (in2*reg_mask);
out3 += (in3*reg_mask);
out4 += (in4*reg_mask);
out5 += (in5*reg_mask);
in0=in1; in1=in2; in2=in3;
in3=in4; in4=in5;
} }
out[row][col]=out0/159;
out[row][col+1]=out1/159;
out[row][col+2]=out2/159;
out[row][col+3]=out3/159;
out[row][col+4]=out4/159;
out[row][col+5]=out5/159; } }
```

Fig. 3 An example, Gaussian Blur algorithm

exposed by applying loop unroll to the innermost iterator. Regarding the loop unroll factor values, they depend on the RF size; the larger the loop unroll factor value, the larger the number of the registers needed. Moreover, the unroll factor values and the number of the variables/registers used, depend on data reuse, as it is efficient to use more variables/registers for the array references that achieve data reuse and less for the references that do not. Thus, the above problems are strongly interdependent and it is not efficient to be addressed separately.

The number of assigned variables/registers is given by the following inequality. I generate mathematical inequalities that give all the efficient RF tile sizes and shapes, according to the number of the available registers. All the schedules with different number of registers than those the proposed inequalities give are not considered, decreasing the search space. In contrast to all the related works, the search space is decreased not by applying exploration and prediction methods but by fully utilizing the RF size.

The register file inequality is given by:

$$0.7 \times RF \leq Iterators + Scalar + Extra + Array_1 + Array_2 + \cdots + Array_n \leq RF \quad (1)$$

where RF is the number of available registers, $Scalar$ is the number of scalar variables, $Extra$ is the number of extra registers needed, i.e., registers for intermediate results and $Iterators$ is the number of the different iterator references exist in the innermost loop body, e.g., in Fig. 3b, ($Iterators = 4$), these are (row, mr, col, mc). $Array_i$ is the number of the variables/registers allocated for the i-th array.

Rule 1 *The $Array_i$ values are found by in Eq. 2 and Rules 2–5. I give Rules 2–5 to allocate registers according to the data reuse.*

$$Array_i = it'_1 \times it'_2 \times \cdots \times it'_n \quad (2)$$

where the integer it'_i are the unroll factor values of the iterators exist in the array's subscript; e.g. for $B(i, j)$ and $C(i, i)$, $Array_B = i' \times j'$ (rectangular tile) and $Array_C = i'$ (diagonal line tile) respectively, where i' and j' are the unroll factors of i, j iterators.

Each subscript equation contributes to the creation of in Eq. (1), i.e., equation i gives $Array_i$ and specifies its expression. To my knowledge, no other work uses a similar approach.

The bound values of the RF inequality (Eq. (1)) are not tight because the output code is C-code and during its compilation (translate the C-code into binary code), the target compiler may not allocate the exact number of desirable variables into registers. Moreover, the 0.7 value has been found experimentally. The number of $Scalar$ and $Extra$ registers are found after the allocation of the array elements into variables/registers, because they depend on the unroll factor values and on the number of iterators being unrolled. The $Extra$ value depends on the target compiler and thus it is found approximately; the bounds of the RF inequality are not tight for this reason too. The goal is to store all the inner loop reused array elements and scalar variables into registers minimizing the number of register spills.

Rules 2–5 modify Eq. 2 according to the data reuse. The array references that do not achieve data reuse do not use more than one register; on the other hand, the array references that achieve data reuse, use as many registers as possible according to the number of the available registers, e.g., in Fig. 3c only one register is used for (in) and four for (out) array. To my knowledge, no other work takes into account data reuse in this way.

Rule 2 *The innermost iterator is never unrolled because data reuse cannot be exposed; if it_i is the innermost iterator, then $it'_i = 1$.*

Proof By unrolling the innermost iterator, the array references-equations which contain it, will change their values in each iteration; this means that (1) a different element is accessed in each iteration and thus a huge number of different registers is needed for these arrays, (2) all these registers are not reused (a different element is accessed in each iteration). Thus, by unrolling the innermost iterator, more registers are needed which do not achieve data reuse; this leads to low RF utilization. □

Rule 3 *The type1 array references which contain all the loop kernel iterators, do not achieve data reuse (each element is accessed once); thus only one register is needed for these arrays, i.e., $Array_i = 1$.*

Proof The subscript equations of these arrays change their values in each iteration vector and thus a different element is fetched in each iteration. □

Let me give an example Gaussian Blur (Fig. 3). Suppose that loop unroll is applied to row, col iterators with unroll factor value equal to 2 (Fig. 3b). Then, all array references are replaced by variables/registers according to the register file size and data reuse (Fig. 3c). Array *out* needs $row' \times col' = 4$ registers (Fig. 3) while array *mask* needs $mr' \times mc' = 1$ register, where row', col', mr', mc' are the row, col, mr, mc unroll factor values, respectively. Although *in* array needs 4 registers according to in Eq. 2, it achieves no data reuse and thus only one register is used (Rule 3). The number of registers needed for each array is given by its subscript equation; these equations give the data access patterns and data reuse. Regarding *mask* array, it is reused four times. Regarding *out* array, all its four references exist in the loop body remain unchanged as the mr, mc change, achieving data reuse. *in* array does not achieve data reuse and thus only one register is used for all the four references. However, if someone study *in* array more carefully, it contains data reuse between consecutive iterations of mc; in Fig. 3d this type of data reuse (Rule 5) is fully exploited by inserting $in1 - in5$ registers. It is important to say that both Fig. 3c, d, achieve a much smaller number of L/S and addressing instructions.

Rule 4 *If there is an array reference (1) containing more than one iterator and one of them is the innermost one and (2) all in Eq.* (1) *iterators which do not exist in this array reference have unroll factor values equal to* 1*, then only one register is needed for this array, i.e. $Array_i = 1$. This gives more than one register file inequalities (it is further explained in the following example).*

Proof When Rule 4 holds, a different array's element is fetched in each iteration vector, as the subscript equation changes its value in each iteration. Thus, no data reuse is achieved and only one register is used. On the contrary, if at least one iterator which do not exist in this array reference is unrolled, common array references occur inside the loop body (e.g., $regC1$ is reused 3 times in Fig. 4); data reuse is achieved in this case and thus an extra RF inequality is created. □

Let me give another example (Fig. 4a). The C array subscript contains i and j iterators. j iterator is the innermost one and thus ($i' \times 1 = 2$) registers are needed for

(a)

```
for (ii=0; ii<60; ii+=10)
 for (jj=0; jj<60; jj+=15)
  for (kk=0; kk<60; kk+=4)
    for (i=ii; i<ii+10; i+=2)
     for (k=kk; k<kk+4; k+=4) {
regA1=A[i][k]; regA2=A[i][k+1];
 regA3=A[i][k+2]; regA4=A[i][k+3];
regA5=A[i+1][k]; regA6=A[i+1][k+1];
regA7=A[i+1][k+2]; regA8=A[i+1][k+3];
    for (j=jj; j<jj+15; j++) {
     regC1=0; regC2=0; regB1=B[k][j];
     regB2=B[k+1][j];
     regB3=B[k+2][j]; regB4=B[k+3][j];
    regC1+=regA1 * regB1;
    regC1+=regA2 * regB2;
    regC1+=regA3 * regB3;
    regC1+=regA4 * regB4;
    regC2+=regA5 * regB1;
    regC2+=regA6 * regB2;
    regC2+=regA7 * regB3;
    regC2+=regA8 * regB4;
C[i][j]+=regC1;
C[i+1][j]+=regC2; }}
```

(b)

```
i0=2; j0=2;
for (jj=0; jj<M; jj+=T1)
  for (i=0; i<N; i+=i0) {
    for (j=jj; j<jj+T1; j+=j0) {
     r1=0;r2=0;r3=0;r4=0;
     for (k=0; k<P; k++) {
  r5=A[i][k]; r6=A[i+1][k];
  r7=B[j][k]; r8=B[j+1][k];
  r1+=r5*r7;
  r2+=r5*r8;
  r3+=r6*r7;
  r4+=r6*r8;
  }
  C[i][j]=r1;
  C[i][j+1]=r2;
  C[i+1][j]=r3;
  C[i+1][j+1]=r4;
  } }
```

Fig. 4 An example, Matrix–Matrix multiplication

this array; however, according to Rule 4, C array needs ($i' \times 1 = 2$) registers if $k' \neq 1$ and 1 register otherwise (if $k' = 1$ then the C array fetches a different element in each iteration vector and thus only one register is needed). Regarding array A, it needs $i' \times k'$ registers while B array needs k' registers, if $i' \neq 1$ and 1 register otherwise. Note that if the i-loop is not unrolled ($i' = 0$), the B and C array elements are not reused and thus there is 1 register for C and 1 for B (Rule 4). The innermost iterator (j) is not unrolled according to Rule 2 (data reuse is decreased in this case).

Moreover, there are cases that data reuse utilization is more complicated as common array elements may be accessed not in each iteration, but in each k iterations, where $k \geq 1$, e.g., Fig. 3d. This holds only for type2 equations (e.g. $ai + bj + c$) where $k = b/a$ is an integer (data reuse is achieved in each k iterations). Data reuse is exploited only when $k = 1$ here (Rule 5) as for larger k values, data reuse is low. For example, in Fig. 3, each time the $mask$ is shifted by one position to the right (mc iterator), 20 elements of in array are reused (reuse between consecutive iterations, i.e., $k = 1$).

Rule 5 *Arrays with type2 subscript equations which have equal coefficient absolute values (e.g. $ai + bj + c$, where $a == \pm b$) fetch identical elements in consecutive iterations; data reuse is exploited by interchanging the registers values in each iteration. An extra RF inequality is produced in this case.*

Proof The arrays of Rule 5 access their elements in patterns. As the innermost iterator changes its value, the elements are accessed in a pattern, i.e. $A[p]$, $A[p + b]$, $A[p + 2 \times b]$ etc. When the outermost iterator changes its value, this pattern is repeated,

shifted by one position to the right ($A[p+b]$, $A[p+2\times b]$, $A[p+3\times b]$, etc.), reusing its elements. This holds for equations with more than two iterators too. □

To exploit data reuse of Rule 5, all the array's registers interchange their values in each iteration, e.g., in Fig. 3d, the ($in0, in1, in2, in3, in4, in5$) variables interchange their values in each iteration.

To sum up, by applying the aforementioned transformations as above, the number of (1) load/store instructions (or equivalent the number of L1 data cache accesses) and (2) addressing instructions, are decreased. The number of load/store instructions is decreased because the reused references are assigned into registers and they reused as many times as the number of available registers indicate. The number of addressing instructions is decreased because the address computations are simplified.

Last but not least, the search space is decreased by orders of magnitude without pruning efficient schedules. Although it is impractical to run all the different schedules (their number is huge) to prove that they are less performance efficient, they all give a much larger number of load/store and addressing instructions. All the schedules that do not belong to in Eq. 1, either use only a few number of registers or a larger number than the available; in the first case, array elements are accessed more times (and also their addresses are computed more times) while in the second case, the register file pressure is high, leading to a large number of register spills and therefore to a large number of L1 data accesses. Moreover, all the schedules that do not satisfy the proposed Rules, do not take into account data reuse and thus several/many registers are wasted leading to a larger number of data accesses.

3.2 Loop tiling and data array layouts

Loop tiling is one of the key loop transformations to speedup data dominant applications. When the accumulated size of the arrays is larger than the cache size, the arrays do not remain in cache and in most cases they are loaded and reloaded many times from the slow main memory, decreasing performance and increasing energy consumption. In order to decrease the number of data accesses, loop tiling is applied, i.e., arrays are partitioned into smaller ones (tiles) in order to remain in cache achieving data locality.

Loop tiling for cache, cache size and associativity, data reuse and data array layouts, strongly depend on each other and thus they are addressed together, as one problem and not separately. The reason follows. Let me give an example, Matrix-Matrix Multiplication algorithm. Many research works as well ATLAS [48] (one of the state of the art high performance libraries) apply loop tiling by taking into account only the cache size, i.e., the accumulated size of three rectangular tiles (one of each matrix) must be smaller or equal to the cache size; however, the elements of these tiles are not written in consecutive main memory locations (the elements of each tile sub-row are written in different main memory locations) and thus they do not use consecutive data cache locations; this means that having a set-associative cache, they cannot simultaneously fit in data cache due to the cache modulo effect. Moreover, even if the tile elements are written in consecutive main memory locations (different data array layout), the three tiles cannot simultaneously fit in data cache if the cache is two-way associative or direct mapped [49,50]. Thus, loop tiling is efficient only when cache size, cache

associativity and data array layouts, are addressed together as one problem and not separately.

For a 2 levels of cache architecture, 1 level of tiling (either for L1 or L2 cache), 2 levels of tiling and no tiling solutions, are applied to all the solutions-schedules that have been produced so far. The optimum number of levels of tiling cannot easily be found since the data locality advantage may be lost by the required insertion of extra L/S and addressing instructions, which degrade performance. The separate memories optimization gives a different schedule for each memory and these schedules cannot coexist, as by refining one, degrading another, e.g., the schedule minimizing the number of L2 data cache accesses and the schedule minimizing the number of main memory accesses cannot coexist; thus, either a sub-optimum schedule for all memories or a (near)-optimum schedule only for one memory can be produced.

As far as the tile sizes are concerned, a cache inequality is produced for each cache memory (Eq. 3), giving all the efficient tile sizes; each inequality contains (1) the tile size of each array and (2) the shape of each array tile. However, partitioning the arrays into smaller ones (tiles) according to the cache size is not enough because tiles may conflict with each other due to the cache modulo effect; to satisfy that tiles remain in cache, first all the tile elements are written in consecutive main memory locations (in order to be loaded in consecutive cache locations), second different array tiles are loaded in different cache ways in order not to conflict with each other and third LRU cache replacement policy is used. All tile sizes and shapes and array layouts different than these the proposed inequalities give are not considered decreasing the search space.

The L1 data cache inequality is given by:

$$assoc - v - (\lfloor assoc/4 \rfloor) \leq \left\lceil \frac{Tile_1}{L_1/assoc} \right\rceil + \cdots + \left\lceil \frac{Tile_n}{L_1/assoc} \right\rceil \leq assoc - v \tag{3}$$

where L_1 is the L1 data cache size and $assoc$ is the data cache associativity (for an 8-way associative data cache, $assoc = 8$). v value is zero when no type3 array exist and one if at least one type3 array exists (it is explained next, Rule 6). $(\lfloor assoc/4 \rfloor)$ gives the number of cache ways that remain unused and defines the lower bound of tile sizes. $Tile_i$ is the tile size of the ith array and it is given by

$$Tile_i = T_1' \times T_2' \times T_n' \times ElementSize \tag{4}$$

where integer T_i' are the tile sizes of the iterators that exist in the array's subscript. $ElementSize$ is the size of each array's element in bytes, e.g., in Fig. 4a, $Tile_A = ii' \times kk' \times 4 = 10 \times 4 \times 4$, $Tile_B = kk' \times jj' \times 4 = 15 \times 4 \times 4$ and $Tile_C = ii' \times jj' \times 4 = 10 \times 15 \times 4$, if A, B and C contain floating point values (4 bytes each).

In Eq. 3 satisfies that no cache way contains more than one array's elements, minimizing the number of cache conflicts. $a1 = \lceil \frac{Tile_1}{L_1/assoc} \rceil$ value is an integer and gives the number of L_1 data cache lines with identical L_1 addresses used for $Tile_1$. An empty cache line is always granted for each different modulo (with respect to the size of the cache) of Tiles memory addresses. For the reminder of this paper I will more

freely say that I use $a1$ cache ways for $Tile1$, $a2$ cache ways for $Tile2$ etc (in other words Tiles are written in separate data cache ways). In the case that $\frac{Tile_i}{L_1/assoc}$ would be used instead of $\lceil \frac{Tile_i}{L_1/assoc} \rceil$, the number of L_1 misses will be much larger because Tiles' cache lines would conflict with each other.

Moreover, in order to the Tiles remain in cache, their elements have to be written in consecutive cache locations and thus to consecutive main memory locations; thus, the data array layouts are changed to tile-wise (if needed), i.e., all array elements are written to main memory in order. To my knowledge, no other work addresses loop tiling by taking into account cache size, cache associativity and the data array layouts for a wide range of algorithms and computer architectures.

In the case that the tile size is very small ($Tile_i \prec (L1/assoc)/10$), neither its layout is changed nor it is inserted in in Eq. 3 (this value has been found experimentally). Moreover, if the tile elements are not written in consecutive main memory locations but the associativity value is large enough to prevent cache conflicts, the data array layout remains unchanged, e.g., consider a 2-d array of size $N \times N$ and a tile of size $4 \times T$, where $T \prec N$; if ($assoc \geq 4$) and (($T \times ElementSize$) $\prec CacheWaySize$), no cache conflict occurs.

Regarding type3 arrays, loop tiling cannot be applied. These arrays contain subscript equations which are not given by a compile time known expression (they depend on the input data). These arrays cannot be partitioned into tiles as their elements are accessed in a 'random' way; this leads to a large number of cache conflicts due to the cache modulo effect (especially for large arrays). To eliminate these conflicts, Rule 6 is introduced.

Rule 6 *For all the type3 arrays, data cache size which equals to the size of one cache way is granted ($v = 1$). In other words, an empty cache line is granted for each different modulo (with respect to the size of the cache) of these arrays memory addresses.*

The search space is decreased even more by computing the best nesting level values of the new tiling iterators.

Statement 1 *The nesting level values of the tiling iterators (produced by applying loop tiling) are found theoretically (no searching is applied). This means that loop interchange is not applied to the new tiling iterators.*

The nesting level values of the new (tiling) iterators are computed. For each different schedule produced by in Eq. (3), I compute the total number of data accesses for all the different nesting level values and the best are selected.

The number of each array's accesses is found by:

$DataAccesses = n \times Tile\_size\_in\_elements \times Num\_of\_tiles$, where n is the number of times each tile is fetched and equals to ($q \times r$), where q is the number of iterations exist above the upper iterator of the array's equation and r is the number of iterations exist between the upper and the lower iterators of the array's equation.

Last but not least, the search space is decreased by orders of magnitude without pruning efficient schedules. Although it is impractical to run all different schedules (their number is huge) to prove that they are less efficient, they all give a much larger number of memory accesses in the whole memory hierarchy. All schedules that do not belong to in Eq. 3, either use only a small portion of cache or a larger than the

available or tile sizes do not use consecutive main memory/cache locations and thus they cannot remain in cache; in the first case, tile sizes are small giving a large number of data accesses and addressing instructions while in the second case, tile sizes are large and tiles cannot remain in cache leading to a much larger number of memory accesses.

4 Experimental results

The experimental results for the proposed methodology, presented in this section, were carried out with Intel Xeon Quad Core E3-1240 v3 general purpose processor and PowerPC 440 embedded processor (Virtex-5 FPGA ML507 Evaluation Platform). Regarding Intel processor, a performance comparison is made over gcc 4.8.4 and Intel icc 2015 compilers; optimization level -O3 was used at all cases. The operating system used is Ubuntu 14.04.

The comparison is done for 7 well-known data dominant linear algebra, image processing and signal processing kernels (PolyBench/C benchmark suite version 3.2 [51]). These are: Matrix-Matrix Multiplication (MMM), Matrix-Vector Multiplication (MVM), Gaussian Blur (5×5 filter), Finite Impulse Response filter (FIR), Sobel operator (Manhattan distance is used instead of Euclidean distance), Jacobi 9-point Stencil and Gaussian Elimination. All the C-codes are single threaded and thus they run on one core only.

First, an evaluation of compilation time/search space is made over iterative compilation (iterative compilation here includes all the transformations shown in Fig. 1.). To the best of my knowledge there is no iterative compilation method including the optimizations presented in this paper with all their parameters; iterative compilation techniques either do not use the transformations presented in this paper at all, or they use some them to some extent [21–23], e.g., loop tiling is applied only for specific tile sizes and levels of tiling and loop unroll is applied only for specific unroll factor values. Normally, iterative compilation methods include transformations with low compilation time such as common subexpression elimination, unreachable code elimination, branch chaining and not compile time expensive transformations such as loop tiling; I show that if the transformations presented in Fig. 1 (including almost all different transformation parameters) are included in iterative compilation, the search space is from 10^{17} up to 10^{29} schedules(for the given input sizes) (Table 1); given that $1sec = 3.17 \times 10^{-8}$ years and supposing that compilation time takes 1 sec, the compilation time is from 10^{9} up to 10^{21} years. On the other hand, the proposed methodology decreases the search space from 10^{1} up to 10^{5} schedules.

The first column of Table 1 gives the overall size of the search space (Fig. 1); the values have been computed by the following equation:

$$\begin{aligned} Schedules = 7! \times (&UnrollFactor^{loops} \times loops! \\ &+ UnrollFactor^{loops} \times Tile1^{loops} \times (2 \times loops)! \times (2 \times matrices) \\ &+ UnrollFactor^{loops} \times Tile2^{loops} \times (2 \times loops)! \times (2 \times matrices) \\ &+ UnrollFactor^{loops} \times Tile1^{loops} \times Tile2^{loops} \times (3 \times loops)! \\ &\times (2 \times matrices)) \end{aligned} \tag{5}$$

Table 1 Evaluation of the search space (compilation time)

	Estimated total search space	Estimated search space of Sect. 3.1	Number of schedules generated by Sect. 3.1	Estimated search space of Sect. 3.2	Number of schedules generated by Sect. 3.2	Total number of schedules generated by Prop. method
MMM (N = 1000)	10^{29}	4.4×10^6	4.2×10^1	10^{20}	8.0×10^3	3.3×10^5
MVM (N = 4200)	10^{22}	2.4×10^4	1×10^1	10^{15}	4.5×10^2	4.6×10^3
Gaussian Blur (N = 1200)	10^{27}	1.4×10^7	1.2×10^2	10^{13}	2.9×10^2	3.6×10^4
FIR (N = 32000, M = 4000)	10^{17}	2.4×10^4	1.6×10^1	10^{10}	3.4×10^1	5.6×10^2
Sobel (N = 1200)	10^{27}	5.2×10^6	1.1×10^2	10^{13}	2.9×10^2	3.1×10^4
Jacobi 9-point Stencil (N = 1200)	10^{27}	5.2×10^6	8.1×10^1	10^{13}	2.9×10^2	2.5×10^4
Gaussian Elimination (N = 1200)	10^{17}	1.5×10^6	6.0×10^0	10^{10}	2.3×10^1	1.4×10^2

The values shown are numbers of schedules/binaries (fixed input sizes)

where $Tile1$ and $Tile2$ are the numbers of different tile sizes for each iterator for the L1 and L2, respectively and $loops$ is the number of the loops. $UnrollFactor$ value is the number of different unroll factor values for each iterator. Finally, $matrices$ is the number of multidimensional arrays and indicates that each multidimensional array uses two different data layouts (the default and the tile-wise). The first, second, third and fourth row of Eq. 5, give the number of the schedules when tiling for (no memory), (L1), (L2) and (L1 and L2) is applied, respectively.

The second and the fourth columns of Table1 give the search space size of the transformations used in Sects. 3.1 and 3.2, respectively. The second column values of Table 1 are given by $(4! \times UnrollFactor^{loops} \times loops!)$. The fourth column values are obtained by $(4! \times Tile1^{loops} \times Tile2^{loops} \times (2 \times loops)! \times (2 \times matrices))$. The other column values have been obtained experimentally. The search space of MMM is the biggest as it contains 3 large loops to which loop tiling has applied. On the other hand, the smallest search space is that of MVM as it contains only two loops and a small input size.

For a fair comparison to be made, $(UnrollFactor = 32)$, since the number of the registers is limited. Moreover, the different tile sizes used here are the following: for MMM $Tile1 = 500$ and $Tile2 = 250$, for Gaussian Blur, Sobel and Jacobi stencil $Tile1 = 600$ and $Tile2 = 300$, for MVM $Tile1 = 2100$ and $Tile2 = 1000$, for FIR $Tile1 = 16000$ (1st iterator), $Tile1 = 1000$ (filter iterator) and $Tile2 = 0$, for Gaussian Elimination, $Tile1 = 600$ and $Tile2 = 0$. Regarding FIR and Gaussian Elimination, $Tile2 = 0$ as their arrays fit in L2 cache and therefore tiling for L2 is useless.

MMM achieves the largest number of schedules and thus the largest compilation time, because it contains three large loops which are eligible to loop tiling. On the other hand, Gaussian Elimination and FIR achieve the smallest number of schedules since their arrays are of small size and they fit in L2 cache (for the given input sizes); this means that no tiling for L2 is applied, decreasing the number of the schedules. As it was expected, the search space is decreased by many orders of magnitude at all the steps of the proposed methodology. As far as Sect. 3.1 is concerned, the second column shows the estimated number of the schedules and the third shows the number of the schedules generated by the proposed methodology; the search is space is decreased from 3 up to 6 orders of magnitude. As far as Sect. 3.2 is concerned, the fourth column shows the estimated number of schedules and the fifth shows the number of the schedules generated by the proposed methodology; the search is space is decreased from 9 up to 17 orders of magnitude. Regarding the overall estimated search space, it is shown at the first column of Table 1; given that $1s = 3.17 \times 10^{-8} years$, the compilation time is from 10^9 up to 10^{21} years. At last, instead of testing all these schedules (which is impractical), the proposed methodology tests only the schedules shown at the last column of Table 1.

Second, an evaluation of performance is made over iterative compilation. Given that (a) there is no iterative compilation method including all the transformations presented in this paper, including all different transformation parameters, (b) the number of different schedules is huge (1st column of Table 1), I evaluated the proposed methodology only with the most performance efficient transformations, i.e., loop tiling, loop unroll and loop interchange. The proposed methodology is compared with (a) one level of

tiling to one loop (best loop and best tile size), (b) one level of tiling to all the loops (best number of levels of tiling and best tile sizes), (c) loop interchange and one level of tiling to all the loops (best iterator nesting level values, best number of levels of tiling and best tile sizes) and (d) loop unroll to one loop and scalar replacement (best loop and best unroll factor value) (Table 2). Given that even the search space of the (a)–(d) above is huge, only one different set of input sizes is used (the input sizes are those shown in Table 1). Moreover, the number of different tile sizes tested here are limited (from 1 up to 2 orders of magnitude smaller than those used to estimate Table 1). Even for a limited input and tile sizes the number of different binaries is large and thus the experimental results of Table 2 took several days.

It is important to say that the (d) set of transformations above, evaluates Sect. 3.1; in (d), one loop is unrolled (best loop and best unroll factor value) and then all the array references that exist in the loop body are replaced by scalar variables. The (d) set of transformations achieves a much higher speedup than the (a)–(c) ones as it decreases the number of both load/store (L/S) and addressing instructions. However, it does not take into account data reuse and the number of registers and this is why the methodology given in Sect. 3.1 performs about 1.25 times faster. As it was expected, by unrolling the innermost loops, performance is not highly increased since the data reuse being achieved is low.

As it was expected, one level of loop tiling is not performance efficient for Gaussian Blur, Sobel and Jacobi Stencil since the locality advantage is lost by the additional addressing (tiling adds more loops) and load/store instructions (there are overlapping array elements which are loaded twice [52]). Regarding Gaussian Elimination, loop tiling is not performance efficient because the loops allowed to be tiled (data dependencies) (a) do not have fixed bound values (data reuse is decreased in each iteration), (b) the upper row of the matrix (which is reused many times) always fits in L1. This is why the methodology given in Sect. 3.1 achieves the best observed speedup for the Gaussian Blur, Sobel, Jacobi Stencil and Gaussian Elimination.

Regarding MMM, MVM and FIR, loop tiling is performance efficient, especially, when it is applied according to Sect. 3.2; Section 3.2 applies loop tiling in a more efficient way (it takes into account data reuse, memory architecture details and data array layouts) giving higher speedup values. Regarding MMM, loop tiling is performance efficient for matrix sizes larger than 90×90 for both processors (both processors contain 32kbyte L1 data cache). At the MVM case, loop tiling is performance efficient for matrix sizes larger than 4096×4096. As far as FIR is concerned, loop tiling is effective for filter sizes larger than 4000.

Loop tiling is more efficient on PowerPC processor because PowerPC contains only one level of cache, making the memory management problem more critical. In contrast to MMM, the MVM and the FIR do not increase their performance by applying loop tiling to more than one loop iterator; this is because MMM contains three very large arrays and one extra iterator, making loop tiling transformation more critical. As far as loop interchange transformation is concerned, it does not increase performance except from MMM in PowerPC (I believe that gcc already applies loop interchange, at least on Intel processor).

Moreover, an evaluation over gcc/icc compilers is made. Icc performs better than gcc, for all algorithms. A large number of different input sizes is tested for each

Table 2 Performance evaluation for fixed input sizes (those of Table 1)

	One level of tiling to one loop (best loop and best tile size)	One level of tiling to all loops (best tile sizes)	Loop interchange and one level of tiling to all loops	Loop unroll to one loop and scalar replacement	Proposed method. Sect. 3.1 only	Proposed method overall
MMM Intel	1.14	1.15	1.15	1.93	2.42	3.83
MMM PowerPC	1.30	1.51	1.56	1.59	2.02	3.12
MVM Intel	1.08	1.08	1.08	2.07	2.46	2.46
MVM PowerPC	1.10	1.10	1.10	1.63	1.87	2.09
Gaussian Blur Intel	1.00	1.00	1.00	1.92	2.40	2.40
Gaussian Blur PowerPC	1.00	1.00	1.00	2.32	2.83	2.83
FIR Intel	1.09	1.09	1.09	1.70	2.05	2.21
FIR PowerPC	1.15	1.15	1.15	1.61	1.98	2.32
Sobel Intel	1.00	1.00	1.00	1.68	2.01	2.01
Sobel PowerPC	1.00	1.00	1.00	1.82	2.25	2.25
Jacobi 9-point Stencil Intel	1.00	1.00	1.00	1.46	1.82	1.82
Jacobi 9-point Stencil PowerPC	1.00	1.00	1.00	1.76	2.13	2.13
Gaussian Elimination Intel	1.00	1.00	1.00	1.33	1.62	1.62
Gaussian Elimination PowerPC	1.00	1.00	1.00	1.52	1.84	1.84

The speedup values refer to the ratio of the benchmark code execution time to the transformed code execution time (gcc compiler is used)

Table 3 Performance comparison over gcc 4.8.4, icc 2015 and PowerPC compilers

Average speedup	MMM	MVM	Gaussian Blur	FIR	Sobel	Jacobi 9-point Stencil	Gaussian Elimination
IntelXeonE3-1240v3							
gcc	3.3	2.5	2.3	2.0	1.9	1.8	1.6
icc	2.5	1.9	2.1	1.6	1.5	1.4	1.3
PowerPC440							
PowerPC compiler	2.6	1.9	2.7	2.0	2.2	1.9	1.8

algorithm and the average speedup value is computed. As the input size increases, the speedup slightly increases; this is because loop tiling becomes more critical in this case. It is important to say that by slightly changing the input size, the output schedule changes. The input sizes are the following. For MMM square matrices of sizes ($64 \times 64 - 4000 \times 4000$) are used while for MVM a square matrix of size ($256 \times 256 - 160{,}000 \times 16{,}000$) is used. For Gaussian Blur, Sobel, Jacobi Stencil and Gaussian Elimination square matrices of sizes ($256 \times 256 - 3000 \times 3000$) are used. Regarding FIR, (array size, filter size) = ((1000, 80) − (128,000, 12,000)). It is important to say that PowerPC is connected to a DDR2 of 256 Mbytes and therefore the input sizes that exceed this value are not tested.

The proposed methodology achieves significant speedup values on both processors (Table 3), as the number of load/store instructions and cache accesses is highly decreased. icc applies more efficient transformations than gcc to the benchmark codes, resulting in faster binary code. It is important to say that although icc generates much faster binaries than gcc regarding the benchmark codes, both compilers give close execution time binaries regarding the proposed methodology output codes; this is because both compilers are not able to apply high level transformations at the latter case.

Regarding Intel processor, gcc and icc compilers generate binary code containing SSE/AVX instructions (auto-vectorization) and thus they use 256-bit/128-bit YMM/XMM registers. However, the proposed methodology does not support auto-vectorization (it is out of the scope of this paper); thus, all the benchmark C-codes that used on Intel, use SSE instrinsics. In this way, both input and output codes contain the SSE instrinsics. As far as the PowerPC processor is concerned, benchmark C-codes do not contain vector instructions.

The best observed speedup for Intel processor is achieved for MMM while the best speedup for PowerPC is achieved for Gaussian Blur and MMM (Table 3). MMM achieves the largest speedup values because it contains three very large 2-d arrays providing data reuse (loop tiling has a larger effect in such cases). The MMM schedules that achieve best performance on Intel include loop tiling for L1 and L3 and tile-wise data array layouts for A and B, when $C = C + A \times B$ (when the input size is large enough). The MMM schedules that achieve best performance on PowerPC include tiling for L1 data cache and tile-wise layouts for the arrays A and B. Moreover, the methodology given in Sect. 3.1, has a very large effect on performance for both processors (Table 1). Regarding MVM (Table 3), speedup of 2.5 and 1.9 is achieved

for Intel and PowerPC, respectively. Loop tiling for cache is not performance efficient for Intel processor (speedup lower than 1%) since the total size of the arrays that achieve data reuse is small (Y and X arrays, when $Y = Y + A \times X$) and they fit in fast L2 cache in most cases. However, PowerPC has one level of data cache only and thus loop tiling is performance efficient for large array sizes only. Moreover, the data array layout of A (when $Y = Y + A \times X$) is not performance efficient to be changed for two reasons (at both processors). First, array A size is very large compared to the X and Y ones and second, A is not reused (each element of A is accessed only once).

Regarding Gaussian Blur, a large speedup value is achieved in both cases. Gaussian Blur contains two 2-d arrays (images) and a 5×5 mask array which is always shifted by one position to the right to the input image. The input image and the mask array achieve data reuse and thus both array elements are assigned into registers. Regarding Intel processor, the speedup values are 2.3 and 2.1 for gcc and icc, respectively, while for PowerPC the highest speedup value is achieved, i.e., 2.7. Loop tiling is performance efficient only for very large input sizes. As far as PowerPC is concerned, it achieves a larger speedup value than Intel because by creating a large loop body, compilers apply scalar replacement to the Gaussian mask elements, decreasing the number of load/store and arithmetical instructions; the decreased number of instructions has a larger effect on the smaller PowerPC processor which contains only one ALU unit.

FIR achieves a significant but not large performance gain (Table 3) because FIR arrays are of small size. In the case that L1 data cache size is smaller than twice the size of the filter array, loop tiling is necessary. This is because the filter array and a part of the input array (they are of the same size) are loaded in each iteration; if L1 is smaller than this value, the filter array cannot remain in L1 and thus it is fetched many times from the upper memory. In this case loop tiling for L1 is performance efficient; loop tiling is not applied for Intel L2/L3 cache memories since the total size of the arrays is smaller here.

In Sobel Operator two 3×3 mask arrays are applied to the input image, shifted by one position to the right. I use the Manhattan instead of Euclidean distance because in the latter case performance highly depends on the Euclidean distance computation (especially on PowerPC); I use if-condition statements instead of $atan()$ functions for the same reason. PowerPC achieves a larger speedup value than Intel because by increasing the loop body size, compiler apply scalar replacement to the Sobel mask elements (Sobel mask elements contain only ones and zeros), decreasing the number of load/store and arithmetical instructions; the decreased number of instructions has a larger effect on the smaller PowerPC processor which contains only one ALU unit.

Regarding Jacobi 9-point Stencil, speedup values are 1.7 and 1.4 over gcc and icc, respectively (Intel processor). As far as PowerPC is concerned, a larger speedup value is achieved (1.9). Data reuse is achieved in each iteration for the six of the nine array elements which are assigned into registers. Three loads occur in each iteration and not nine, since six registers interchange their values as in (Fig. 3d). Regarding Intel processor, data reuse is achieved by computing more than one result (from vertical dimension) in each iteration.

Finally, Gaussian Elimination achieves the lowest speedup gain on both processors because of the data dependencies (the proposed methodology can be applied partially).

Loop tiling is not performance efficient on both processors since the upper row of the matrix which achieves data reuse, fits in the L1 data cache at all cases.

5 Conclusions

In this paper, six compiler optimizations are addressed together as one problem and not separately, for a wide range of algorithms and computer architectures. New methods for loop unroll and loop tiling are given which take into account the custom SW characteristics and the HW architecture details. In this way, the search space is decreased by many orders of magnitude and thus it is capable to be searched.

References

1. Triantafyllis S, Vachharajani M, Vachharajani N, August DI (2003) Compiler optimization-space exploration. In: Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization, CGO '03. IEEE Computer Society, Washington, DC, pp 204–215. http://dl.acm.org/citation.cfm?id=776261.776284
2. Cooper KD, Subramanian D, Torczon L (2001) Adaptive optimizing compilers for the 21st Century. J Supercomput 23:2002
3. Kisuki T, Knijnenburg PMW, O'Boyle MFP, Bodin F, Wijshoff HAG (1999) A feasibility study in iterative compilation. In: Proceedings of the Second International Symposium on High Performance Computing, ISHPC '99. Springer, London, UK, pp 121–132. http://dl.acm.org/citation.cfm?id=646347.690219
4. Kulkarni PA, Whalley DB, Tyson GS, Davidson JW (2009) Practical exhaustive optimization phase order exploration and evaluation. ACM Trans Archit Code Optim 6(1):1–36. doi:10.1145/1509864.1509865
5. Kulkarni P, Hines S, Hiser J, Whalley D, Davidson J, Jones D (2004) Fast searches for effective optimization phase sequences. SIGPLAN Not 39(6):171–182. doi:10.1145/996893.996863
6. Park E, Kulkarni S, Cavazos J (2011) An evaluation of different modeling techniques for iterative compilation. In: Proceedings of the 14th international conference on Compilers, architectures and synthesis for embedded systems, CASES '11. ACM, New York, NY, pp 65–74. doi:10.1145/2038698.2038711
7. Monsifrot A, Bodin F, Quiniou R (2002) A machine learning approach to automatic production of compiler heuristics. In: Proceedings of the 10th International Conference on Artificial Intelligence: Methodology, Systems, and Applications, AIMSA '02. Springer, London, pp 41–50. http://dl.acm.org/citation.cfm?id=646053.677574
8. Stephenson M, Amarasinghe S, Martin M, O'Reilly UM (2003) Meta optimization: improving compiler heuristics with machine learning. SIGPLAN Not 38(5):77–90. doi:10.1145/780822.781141
9. Tartara M, Crespi Reghizzi S (2013) Continuous learning of compiler heuristics. ACM Trans Archit Code Optim 9(4):46:1–46:25. doi:10.1145/2400682.2400705
10. Agakov F, Bonilla E, Cavazos J, Franke B, Fursin G, O'Boyle MFP, Thomson J, Toussaint M, Williams CKI (2006) Using machine learning to focus iterative optimization. In: Proceedings of the International Symposium on Code Generation and Optimization, CGO '06. IEEE Computer Society, Washington, DC, pp 295–305. doi:10.1109/CGO.2006.37
11. Almagor L, Cooper KD, Grosul A, Harvey TJ, Reeves SW, Subramanian D, Torczon L, Waterman T (2004) Finding effective compilation sequences. SIGPLAN Not 39(7):231–239. doi:10.1145/998300.997196
12. Cooper KD, Schielke PJ, Subramanian D (1999) Optimizing for reduced code space using genetic algorithms. SIGPLAN Not 34(7):1–9. doi:10.1145/315253.314414
13. Cooper KD, Grosul A, Harvey TJ, Reeves S, Subramanian D, Torczon L, Waterman T (2005) ACME: adaptive compilation made efficient. SIGPLAN Not 40(7):69–77. doi:10.1145/1070891.1065921

14. Cooper KD, Grosul A, Harvey TJ, Reeves S, Subramanian D, Torczon L, Waterman T (2006) Exploring the structure of the space of compilation sequences using randomized search algorithms. J Supercomput 36(2):135–151. doi:10.1007/s11227-006-7954-5
15. Kulkarni PA, Whalley DB, Tyson GS (2007) Evaluating heuristic optimization phase order search algorithms. In: Proceedings of the International Symposium on Code Generation and Optimization, CGO '07. IEEE Computer Society, Washington, DC, pp 157–169. doi:10.1109/CGO.2007.9
16. Chen C, Chame J, Hall M (2005) Combining models and guided empirical search to optimize for multiple levels of the memory hierarchy. In: Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO), pp 111–122. doi:10.1109/CGO.2005.10
17. Haneda M, Khnijnenburg PMW, Wijshoff HAG (2005) Automatic selection of compiler options using non-parametric inferential statistics. In: Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques, PACT '05. IEEE Computer Society, Washington, DC, pp 123–132. doi:10.1109/PACT.2005.9
18. Cavazos J, Fursin G, Agakov F, Bonilla E, O'Boyle MFP, Temam O (2007) Rapidly selecting good compiler optimizations using performance counters. In: Proceedings of the International Symposium on Code Generation and Optimization, CGO '07. IEEE Computer Society, Washington, DC, pp 185–197. doi:10.1109/CGO.2007.32
19. de Mesmay F, Voronenko Y, Püschel M (2010) Offline library adaptation using automatically generated heuristics. In: International Parallel and Distributed Processing Symposium (IPDPS), pp 1–10
20. Dubach C, Jones TM, Bonilla EV, Fursin G, O'Boyle MFP (2009) Portable compiler optimisation across embedded programs and microarchitectures using machine learning. In: Proceedings of the 42Nd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 42. ACM, New York, NY, pp 78–88. doi:10.1145/1669112.1669124
21. Knijnenburg PMW, Kisuki T, Gallivan K, O'Boyle MFP (2004) The effect of cache models on iterative compilation for combined tiling and unrolling. J CCPE 16(2–3):247–270
22. Kim D, Renganarayanan L, Rostron D, Rajopadhye S, Strout MM (2007) Multi-level Tiling: M for the Price of One. In: Proceedings of the 2007 ACM/IEEE Conference on Supercomputing, SC '07. ACM, New York, NY, pp 51:1–51:12. doi:10.1145/1362622.1362691
23. Renganarayanan L, Kim D, Rajopadhye S, Strout MM (2007) Parameterized tiled loops for free. SIGPLAN Not 42(6):405–414. doi:10.1145/1273442.1250780
24. Fursin G, O'Boyle MFP, Knijnenburg PMW (2002) Evaluating iterative compilation. In: Languages and Compilers for Parallel Computing, 15th Workshop, (LCPC), Revised Papers. College Park, MD, , pp 362–376
25. Leather H, Bonilla E, O'Boyle M (2009) Automatic feature generation for machine learning based optimizing compilation. In: Proceedings of the 7th Annual IEEE/ACM International Symposium on Code Generation and Optimization, CGO '09. IEEE Computer Society, Washington, DC, pp 81–91. doi:10.1109/CGO.2009.21
26. Stephenson M, Amarasinghe S (2005) Predicting Unroll Factors Using Supervised Classification. In: Proceedings of the International Symposium on Code Generation and Optimization, CGO '05. IEEE Computer Society, Washington, DC, pp 123–134. doi:10.1109/CGO.2005.29
27. Park E, Pouche LN, Cavazos J, Cohen A, Sadayappan P (2011) Predictive Modeling in a Polyhedral Optimization Space. In: Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization, CGO '11. IEEE Computer Society, Washington, DC, pp 119–129. http://dl.acm.org/citation.cfm?id=2190025.2190059
28. Jantz MR, Kulkarni PA (2013) Exploiting phase inter-dependencies for faster iterative compiler optimization phase order searches. In: Proceedings of the 2013 International Conference on Compilers, Architectures and Synthesis for Embedded Systems, CASES '13. IEEE Press, Piscataway, NJ, pp 7:1–7:10. http://dl.acm.org/citation.cfm?id=2555729.2555736
29. Kulkarni S, Cavazos J (2012) Mitigating the compiler optimization phase-ordering problem using machine learning. SIGPLAN Not 47(10):147–162. doi:10.1145/2398857.2384628
30. Parello D, Temam O, Cohen A, Verdun JM (2004) Towards a Systematic, Pragmatic and Architecture-Aware Program Optimization Process for Complex Processors. In: Proceedings of the 2004 ACM/IEEE Conference on Supercomputing, SC '04. IEEE Computer Society, Washington, DC, p 15. doi:10.1109/SC.2004.61
31. Rong H, Douillet A, Gao GR (2005) Register allocation for software pipelined multi-dimensional loops. SIGPLAN Not 40(6):154–167. doi:10.1145/1064978.1065030

32. Hack S, Goos G (2006) Optimal register allocation for SSA-form Programs in polynomial time. Inf Process Lett 98(4):150–155. doi:10.1016/j.ipl.2006.01.008
33. Nagarakatte SG, Govindarajan R (2007) Register Allocation and Optimal Spill Code Scheduling in Software Pipelined Loops Using 0-1 Integer Linear Programming Formulation. In: Proceedings of the 16th International Conference on Compiler Construction, CC'07. Springer, Berlin, Heidelberg, pp 126–140. http://dl.acm.org/citation.cfm?id=1759937.1759949
34. Sarkar V, Barik R (2007) Extended Linear Scan: An Alternate Foundation for Global Register Allocation. Compiler Construction, 16th International Conference. Proceedings. Braga, Portugal, pp 141–155
35. Smith MD, Ramsey N, Holloway G (2004) A generalized algorithm for graph-coloring register allocation. SIGPLAN Not 39(6):277–288. doi:10.1145/996893.996875
36. Wang L, Yang X, Xue J, Deng Y, Yan X, Tang T, Nguyen QH (2008) Optimizing scientific application loops on stream processors. SIGPLAN Not 43(7):161–170. doi:10.1145/1379023.1375679
37. Baradaran N, Diniz PC A register allocation algorithm in the presence of scalar replacement for fine-grain configurable architectures. arXiv:0710.4702
38. Lozano RC, Schulte C Survey on Combinatorial Register Allocation and Instruction Scheduling. arXiv:1409.7628
39. Sherwood T, Calder B, Emer J (1999) Reducing cache misses using hardware and software page placement. In: Proceedings of the 13th international conference on Supercomputing, ICS '99. ACM, New York, NY, pp 155–164. doi:10.1145/305138.305189
40. Cacaval C, Padua DA (2003) Estimating cache misses and locality using stack distances. In: Proceedings of the 17th annual international conference on Supercomputing, ICS '03. ACM, New York, NY, pp 150–159. doi:10.1145/782814.782836
41. Ghosh S, Martonosi M, Malik S (1997) Cache miss equations: an analytical representation of cache misses. In: Proceedings of the 11th international conference on Supercomputing, ICS '97. ACM, New York, NY, pp 317–324. doi:10.1145/263580.263657
42. Song F, Moore S, Dongarra J (2007) L2 cache modeling for scientific applications on chip multiprocessors. In: Parallel Processing, International Conference on 51. doi:10.1109/ICPP.2007.52
43. Hankins RA, Patel JM (2003) Data morphing: an adaptive, cache-conscious storage technique. In: Proceedings of the 29th international conference on Very large data bases—vol 29, VLDB '2003, VLDB Endowment, 2003, pp 417–428. http://dl.acm.org/citation.cfm?id=1315451.1315488
44. Franz M, Kistler T (1998) Splitting data objects to increase cache utilization. Department of Information and Computer Science University of California, Tech. rep
45. Rubin S, Bodík R, Chilimbi T (2002) An efficient profile-analysis framework for data-layout optimizations. SIGPLAN Not 37:140–153. doi:10.1145/565816.503287
46. Chang SK (2003) Data structures and algorithms, vol 13 of series on software engineering and knowledge engineering. World Scientific, Singapore
47. Kelefouras V, Kritikakou A, Goutis C (2015) A methodology for speeding up loop kernels by exploiting the software information and the memory architecture. Comput Lang Syst Struct 41:21–41. http://dblp.uni-trier.de/db/journals/cl/cl41.html
48. Whaley RC, Petitet A, Dongarra JJ (2001) Automated Empirical Optimization of Software and the ATLAS Project. Parallel Comput 27(1–2):3–35
49. Kelefouras V, Kritikakou A, Mporas I, Vasileios K (2016) A high-performance matrix-matrix multiplication methodology for CPU and gpu architectures. J Supercomput 72(3):804–844. doi:10.1007/s11227-015-1613-7
50. Kelefouras VI, Kritikakou A, Papadima E, Goutis CE (2015) A methodology for speeding up matrix vector multiplication for single/multi-core architectures. J Supercomput 71(7):2644–2667. http://dblp.uni-trier.de/db/journals/tjs/tjs71.html
51. O. S. University, Polybench/c benchmark suite (2012). http://web.cs.ucla.edu/~pouchet/software/polybench/
52. Kelefouras VI, Kritikakou A, Goutis C (2014) A methodology for speeding up edge and line detection algorithms focusing on memory architecture utilization. Supercomput Springer. doi:10.1007/s11227-013-1049-x