Automatic On-chip Memory Minimization for Data Reuse

Qiang Liu*, George A. Constantinides*, Konstantinos Masselos† and Peter Y.K. Cheung*
*Imperial College, London SW7 2BT, U.K.
†University of Peloponnese, Tripolis, Greece
{qiang.liu2, g.constantinides, k.masselos, p.cheung}@imperial.ac.uk

Abstract

FPGA-based computing engines have become a promising option for the implementation of computationally intensive applications due to high flexibility and parallelism.

However, one of the main obstacles to overcome when trying to accelerate an application on an FPGA is the bottleneck in off-chip communication, typically to large memories. Often it is known at compile-time that the same data item is accessed many times, and as a result can be loaded once from large off-chip RAM onto scarce on-chip RAM, alleviating this bottleneck.

This paper addresses how to automatically derive an address mapping that reduces the size of the required on-chip memory for a given memory access pattern. Experimental results demonstrate that, in practice, our approach reduces on-chip storage requirements to the minimum, corresponding to a reduction in on-chip memory size of up to $40\times$ (average $10\times$) for some benchmarks compared to a naïve approach. At the same time, no clock period penalty or increase in control logic area compared to this approach is observed for these benchmarks.

1. Introduction

Data storage and transfer issues heavily affect the area, power, and performance of custom circuit implementations in data dominated application domains such as digital signal processing and scientific computing. The reduction of the number of accesses to large off-chip memories has thus become a key task in system design.

Data reuse, *i.e.* the access of data more than once during algorithm execution, presents opportunities for optimization, and has been given attention in recent years [3]. By keeping frequently used data in embedded FPGA RAMs, execution time and power consumption can be reduced significantly [11]. When the loads and stores to on-chip RAM are determined statically at compile time, such an on-chip memory is known as a scratch-pad memory (SPM) [8].

Data dominated applications like full search motion vector estimation (partially shown in Fig. 1 (a)) usually operate on large amounts of data with regular and predictable memory access patterns. For this example, consider introducing two new arrays that will be mapped to on-chip embedded RAM, *RLc* and *RLp*, to store a subset of elements of the large off-chip arrays *current_frame* and *previous_frame*, respectively. We can choose a loop level, say inside the y loop, to load these arrays with all data accessed in the innermost loop. We may then replace the accesses to the *current_frame* and *previous_frame* arrays with accesses to the new arrays (see Fig. 1 (b)). This is a potentially helpful code transformation, since in the original code the same element of the original arrays may be accessed several times within a single iteration of the y loop, whereas it will only be accessed once in the transformed code. For this example, with typical image parameters, we can obtain a reduction of the number of accesses to the external memories by 16 times. This reduction can lead to large improvements in performance, opening avenues for parallelism, and also reduce the power consumption of the system, at the cost of area overheads as shown in [11]. One of the purposes of this work is to reduce such area overheads.

Clearly some details of this approach are missing before it can be automated within a hardware compilation system. The addressing functions f_c and f_p used to address the two new arrays are left unspecified in this figure, as is the mechanism for loading only the required elements of the two external arrays into the on-chip memory. It is these issues that are addressed in this paper.

Specficially, we propose a mathematical approach for solving the problem of automatically generating small-sized on-chip data reuse arrays, and the corresponding addressing functions required. The main contributions of this paper are therefore:

- a mathematical formalization of the problem of generating small on-chip memories for reused data,
- an algorithm to automatically generate array indexing expressions based on ‘modular mappings’,

- an algorithm to compute the 'right inverse' of such mappings, which is used to transfer data to the data reuse arrays for initialization,
- an application of the proposed technique to compilation of several video processing kernels, resulting in the provably minimal memory sizes for the arrays without clock period or area degradation.

Our approach is fully automated and can be integrated into a hardware compiler.

The rest of the paper is organized as follows. Section 3 states the targeted problem. Section 4 formulates the characterization of data reuse. Our proposed approach is described in Section 5. Section 6 shows the benefit of our approach for real benchmarks, and Section 7 concludes the paper. Throughout the paper, the approach is both described in general and illustrated on a single simple example.

2 Background

There is a wide body of research aimed at improving the cache performance in embedded systems. However, scratch-pad memory (SPM) is preferable to caching when memory access patterns can be statically analyzed at compile time. Our work focuses on data reuse exploration for SPM in configurable systems.

The size of arrays storing reused data have been explicitly discussed in only a few previous works. Kandemir *et al.* [8] propose a compilation approach to explore data reuse in a SPM system. There, the address function is derived directly from the original affine addressing function by eliminating the outer loop indices [7]. For example, we may derive addressing functions f_c and f_p for Fig. 1(b) by eliminating the loop indices x and y from the original addressing functions in Fig. 1(a), resulting in $f_c(i,j,k,l) = Mk + l$ and $f_p(i,j,k,l) = M(i+k) + j + l$. The address function obtained in this way we call the *directly derived mapping*, and we compare our proposed approach to this scheme (to our knowledge, this is the only pre-existing scheme for automatically deriving a valid addressing function). The directly derived mapping has a significant disadvantage: it may result in 'holes' in the on-chip memory arrays, where no data is stored, increasing significantly the memory requirements, as will be shown in this paper.

In the wider context of memory subsystem design for a scheduled program, several memory size reduction approaches have been proposed. Verbauwhede *et al.* [14] start with a scheduled program in an applicative language and estimate the memory size by computing the maximum number of live elements of an array at the same time. Catthoor *et al.* [3] optimize the storage order of data to reuse memory. For their 'intra-array storage optimization', a promising canonical linearization of the array index expression is chosen in terms of the minimal maximum distance between two addresses occupied by the array during its lifetime. This minimum is used as a modulus to map the array into memory under a single dimensional modular mapping [10]. Balasa *et al.* [1] estimate the memory size based on a data flow graph built by an accurate dependence analysis model before scheduling. All these approaches allow the reduction of memory size for an array compared to the original code by taking advantage of scheduled production and consumption times of data items (or a partial order on such times in [1]) under the assumption of affine loop nests and affine indexing expressions in the original code.

Darte *et al.* [4] have recently generalised this prior work, and provided a mathematical framework to study the storage space reuse problem for a scheduled program. The aim here is to build a modular mapping, *i.e.* a mapping of the form $f(x) = Gx \bmod s$ for some matrix G and some vector s, from the (potentially multi-dimensional) original array index x to a new array index, which can reuse memory. Darte shows that such a mapping is uniquely determined by the lattice[1] that forms the kernel[2] of the modular mapping, and hence focuses attention on finding a lattice with good properties. The central mathematical problem involves finding a lattice which only intersects a given zero-symmetric polytope[3] at the zero vector [4]. Here, each integral point within the zero-symmetric polytope represents an index difference between two array elements which cannot be stored in the same memory location. While considerably generalising the previous work, [4] focuses on memory reuse without considering data reuse, and has not been applied to hardware synthesis. The technique in this paper generalises that in [4] to consider data reuse, applies the resulting algorithm to hardware compilation, and demonstrates the resulting significant reduction in on-chip memory size for real benchmark circuits.

3. Motivation and Problem Definition

The proposed data reuse approach targets an affine n-level loop nest surrounding each array access. On-chip 'data reuse arrays' can be allocated to each reference to an array. We assume that each array index in the original code consists of a (potentially multi-dimensional) constant-offset modular mapping of the enclosing loop indices. We note in passing that this generalises the standard affine assumption, as $f(x) = Gx \bmod s + c$ is identical to $f(x) = Gx + c$ when s is sufficiently large. For the remainder of this paper,

[1]An n-dimensional lattice is the set of integer linear combinations of n linearly independent vectors in n dimensional space.

[2]The kernel of a function is the set of all elements mapped to zero by that function.

[3]A polytope is a finite region of n-dimensional space enclosed by a finite number of hyperplanes.

```
For (x=0; x<N/B; x++)
  For (y=0; y<M/B; y++)
   For (i=-P; i<P+1; i++)
    For (j=-P; j<P+1; j++)
     For (k=0; k<B; k++)
      For (l=0; l<B; l++)
      {
        …
        …=current_frame [(B*x+k)*M + B*y+l];

        if (!((B*x+i+k<0) || (B*x+i+k>N-1) ||
           (B*y+j+l<0) || ((B*y+j+l>M-1)))
        …=previous_frame[(B*x+i+k)*M + B*y+j+l];
        ...
      }
                    (a)
```

```
For (x=0; x<N/B; x++)
  For (y=0; y<M/B; y++)
  {
   /*initialize the reuse level arrays RLc, RLp*/
   Forall ...
   {
    RLc[ ] = current_frame[ ];
    RLp[ ] = previous_frame[ ];
   }
   For (i=-P; i<P+1; i++)
    For (j=-P; j<P+1; j++)
     For (k=0; k<B; k++)
      For (l=0; l<B; l++)
      {
        …
        …=RLc [ fc(i,j,k,l) ];

        if (!((B*x+i+k<0) || (B*x+i+k>N-1) ||
           (B*y+j+l<0) || ((B*y+j+l>M-1)))
        …=RLp [ fp(i,j,k,l) ];
        ...
      }
  }
                    (b)
```

Figure 1. The full search motion estimation algorithm. (a) The original code, (b) The code with data reuse arrays.

```
For i = ... to ...
  For j = ... to ...
    For l = ... to ...
      For k = ... to ...
        ...
        ... = A [Qu + Fx + c];

          (a)
```

```
For i = ... to ...
    For j = ... to ...
      …
     /*initialize the reuse level array*/
     Forall y ...
        $RL_A$ [Hy + c] = A [Qu + Hy + c];

     For l = ... to ...
       For k = ... to ...
         …= $RL_A$ [Fx + c];
           (b)
```

```
For i = ... to ...
     For j = ... to ...
       ...
     /*initialize the reuse level array*/
      Forall y ...
        $RL_A$ [y] = A [Qu + h(y)];

      For l = ... to ...
        For k = ... to ...
          …= $RL_A$ [g(x)];
           (c)
```

```
For m=0 to 10
  For i=0 to 1
    For j=0 to 1
      For k=0 to 5

         …= A [4m + 50i + j + k];

              (d)
```

```
For m=0 to 10
{
    /* initialize the reuse level array. */
    For  $y_1$=0 to 1
      For $y_2$ =0 to 6
        $RL_A$ [$50y_1 + y_2$] = A [$4m + 50y_1 + y_2$];

     For i=0 to 1
       For j=0 to 1
         For k=0 to 5
               …= $RL_A$ [50i + j + k];
}
                  (e)
```

Figure 2. (a-c) General loop structure, and (d-e) a simple example. (a) Original program, (b) Program with directly derived data reuse array, (c) Program with a generalised data reuse array, (d) Original example program, (e) Example with a directly derived mapping.

we shall use the more common affine form for clarity of exposition, but the procedure is equally valid in the modular case, *mutatis mutandis*.

A general such loop structure is shown in Fig. 2 (a). Note that the loop bounds need not be constant - they may be any affine function of the outer loop iterators, and that Q, F, and c are general integral matrices and a vector, respectively. The original array A is accessed inside these loops with an indexing expression $E(u,x) = Qu + Fx + c$, where $u = (i,j)^T$ and $x = (l,k)^T$ are an outer loop index vector collecting those iteration indices outside the scope of the reuse array, and a inner loop index vector collecting those iteration indices inside the scope of the reuse array, respectively. This notation captures potentially multi-dimensional arrays in a standard manner by allowing Q, F, and c to have multiple rows. Thus, for example, a 'C-style' expression $A[10i + 2j][5j]$ is written as $A[\begin{pmatrix} 10 & 2 \\ 0 & 5 \end{pmatrix} \begin{pmatrix} i \\ j \end{pmatrix}]$.

Let us say we wish to introduce a reuse array RL_A for the loop j to store repeatedly used data in local small memory in order to reduce accesses to large memory. The array RL_A is initialized inside the loop with iterator j and is accessed inside the inner loops with an indexing function. *Our primary goal is to automatically derive this indexing function.* As discussed in the background, a simple way, shown in Fig. 2 (b), is to derive this function directly from the original indexing function $E(u,x) = Qu + Fx + c$ by eliminating the outer loop indices, resulting in $Fx + c$. Meanwhile, a piece of code is generated to transfer each datum from the original array with an array index $Qu + Hy + c$ to the reuse array only once, where $Hy + c$ could be seen as an image of the mapping $Fx + c$.

In some cases such a *directly generated* address function requires memory space much larger than the number of distinct buffered data. To illustrate this behavior, a simple example is shown in Figs. 2 (d) and (e). The original code in Fig. 2 (d) consists of four regularly nested loops, and exhibits data reuse (for example iteration $m = i = j = 0, k = 1$ and iteration $m = i = k = 0, j = 1$ read the same array element). In Fig. 2 (e) a data reuse array is introduced for the outer-most loop and the address function for it is derived by eliminating the outer-most loop index m. The number of memory elements needed by this choice of indexing expression is 56, while the actual number of buffered data can be seen from a full unrolling, to be only 14. This means that 75% of memory elements are not accessed and are wasted. For real benchmarks, the waste of memory is often considerable, as will be shown in Section 6.

We thus want to derive a new indexing function $g(x)$ in Fig. 2 (c) for the data reuse array, which can reduce the required memory space and control the memory waste within a limited range, but that still preserves the essential characteristics required of a *valid* indexing expression. Moreover, we wish to derive an array index $Qu + h(y)$ for loading buffered elements from the original array. This problem is simple to state, but turns out to have a relatively complex mathematical structure, which we shall present, along with one method for solving the problem.

4. Characterization of the data reuse problem

In this section, we derive a mathematical characterisation of the data reuse, and derive a necessary and sufficient condition for a given indexing expression $g(x)$ to be valid.

We may start with the directly derived indexing expression for the data reuse array RL_A, $f(x) = Fx + c$ which, as illustrated above, may need a larger memory space than that strictly necessary. Our goal is, from this starting point, to obtain an alternative mapping $g(x)$ that reduces the on-chip memory size.

By analyzing the loop structure, we can define the iteration space, $\mathcal{IS}$ of the inner loop nest, where each element corresponds to a iteration vector. For the example code, $\mathcal{IS} = \{(i,j,k)^T \mid 0 \le i,j \le 1,\ 0 \le k \le 5,\ i,j,k \in \mathbb{Z}\}$. To preserve the data reuse, if $f(x) = f(y)$, where $x, y \in \mathcal{IS}$, *i.e.* the same datum is read in iteration x and in iteration y, then we require $g(x) = g(y)$. Since, in our case, $f(x) = Fx + c$, we may write $f(x) = f(y) \Leftrightarrow F(x - y) = 0$. Thus for the remainder of this paper, we work with the set of iteration *differences* $D = \{d \mid d = x - y,\ x, y \in \mathcal{IS}\}$, which is equal to the Minkowski (set theoretic) sum of $\mathcal{IS}$ with $(-\mathcal{IS})$, $\mathcal{IS} \oplus (-\mathcal{IS})$, and can be characterized as the set of all integer points within a 0-symmetric polytope K. For the example, $D = \{(d_1, d_2, d_3) \mid -1 \le d_1, d_2 \le 1, -5 \le d_3 \le 5\}$, and its enclosing polytope K is illustrated in Fig. 3 (a).

Working with the polytope K, we can define a set P, which characterizes the data reuse implicit in the original memory mapping, $P = \{x \mid Fx = 0, x \in D\}$, *i.e.* that part of the *kernel* of the mapping that is also present in the difference set, $P = ker(F) \cap D$. Note that each element in this set corresponds to a difference in two iteration vectors that access the same array element, precisely capturing the data reuse present.

Our task is to construct a new mapping $g(x)$ that preserves the data reuse present in the directly derived mapping $f(x)$, while maintaining correctness of the code by never mapping two iteration vectors that *do not* re-use the same data element to the same array element.

Definition 1 A mapping $g(x)$ is *valid* for the data reuse set P, the iteration difference set D, and the directly derived mapping $f(x) = Fx + c$ iff it satisfies two conditions:
(i) preservation of existing data-reuse: $\forall x\ (x \in P \Rightarrow g(x) = 0)$
(ii) unique indices within the iteration space: $\forall x\ (g(x) = 0 \wedge Fx \neq 0 \Rightarrow x \notin D)$.

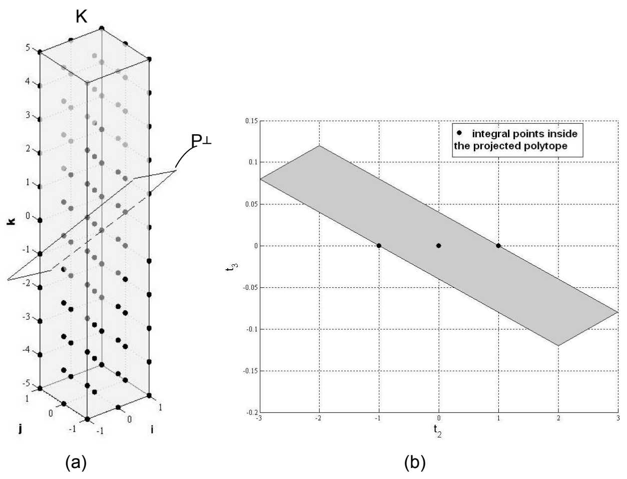


Figure 3. (a) The original polytope K, where dots illustrate integer points of the difference set D, (b) Projection of K onto the basis vectors a_1 and a_2

The first condition captures the fact that if two iterations of the original code access the same memory element, then the corresponding iterations of the transformed code should also access the same memory element. The second condition captures the fact that we can allow the constructed function to map two vectors to the same memory element when the original mapping did not, only if at least one of these iterations is outside the iteration space (and therefore never executed in practice). This second condition can be exploited to 'crunch down' the memory requirement.

The remainder of this paper illustrates how such a mapping can be constructed while aiming for a minimal memory requirement.

5. Proposed Algorithm

As for the original mapping function $f(x)$, we consider the class of *modular mappings* for the potential indexing functions $g(x)$, *i.e.* $g(x) = Gx \bmod s$, where G is an integral matrix and s is a modulus vector [10]. Note that the directly derived mapping from previous work is just a special case of a modular mapping, and thus our technique automatically generates the former, if no advantage can be found from the added flexibility given by our proposed scheme (such cases will be illustrated in Section 6).

The proposed approach consists of a number of steps summarized in Fig. 4, and results in a valid modular mapping for the storage space reduction of the data reuse arrays. There are already several existing algorithms for the first step, the Minkowski (set theoretic) sum [9]; as a result, we focus our discussion on the remaining novel steps.

Parameters: the iteration space *IS* and the linear function *F* for the reuse level array.
1. Use Minkowski sum of *IS* to form the difference polytope *K*
2. Compute the Hermite normal form [12] of *F* to obtain a basis for the kernel of *F*
3. Use Fourier-Motzkin elimination to remove ''redundant'' vectors obtaining an irredundant basis for the reuse characterization set, *P*
4. Find the orthogonal subspace of *P*
5. Project the polytope *K* onto the orthogonal subspace
6. Build a basis for the lattice consisting of the kernel of the required modular mapping
7. Construct a modular mapping *Gx* mod *s* from the lattice
8. Find an inverse of the modular mapping

Figure 4. A summary of the approach.

5.1. Data reuse characterization

As discussed above, the set $P = \ker(F) \cap D$ characterizes the data reuse. We obtain this set in a parametric fashion, by first obtaining a basis $(a_1, a_2, \ldots, a_k)$ for ker(F), where $a_i \in \mathbb{Z}^n$ and then removing any unnecessary vectors from this basis. A basis for a linear integral mapping can be found by computing the so-called *Hermite normal form* of the matrix F [12], as follows.

Given a feasible set of linear equations $Fx = \mathbf{0}$ in integer variables x, where F is a $q \times n$ integral matrix, $q \leq n$, and x is an n-dimensional vector, we can find a unimodular matrix[4] V such that $FV = [H\ \mathbf{0}]$. The $q \times m$ matrix H is known as the Hermite normal form of F, and provides a basis for the set of integer solutions to $Fx = \mathbf{0}$, given in (1), where $\mathbf{0}_{p,b}$ is the $p \times b$ matrix of zeros and I_p is the $p \times p$ identity matrix.

$$(a_0, a_1, ..., a_{n-q}) = V \begin{pmatrix} \mathbf{0}_{q,1} & \mathbf{0}_{q,n-q} \\ \mathbf{0}_{n-q,1} & I_{n-q} \end{pmatrix} \qquad (1)$$

Any solution to $Fx = \mathbf{0}$ is a linear combination of these basis vectors with integral coefficients [12]. We use this procedure to obtain a basis for $\ker(F)$. Algorithms for Hermite normal are discussed in [12] and are built into Matlab and Maple.

For the example code from Fig. 2(d), our goal is to characterize the integral solutions of the equation $50x_1 + x_2 + x_3 = 0$. Writing this as $Fx = \mathbf{0}$, gives $F = (50, 1, 1)$, $x = (x_1, x_2, x_3)^T$. The solution produced by the method above is given in (2), where t_1 and t_2 are any integers.

$$x = \begin{pmatrix} 0 & -1 \\ 1 & 25 \\ -1 & 25 \end{pmatrix} \begin{pmatrix} t_1 \\ t_2 \end{pmatrix} \qquad (2)$$

Thus the two basis vectors $a_1 = (0, 1, -1)^T$ and $a_2 = (-1, 25, 25)^T$ fully characterize the data reuse present at

[4]A unimodular matrix is a square matrix with determinant ± 1

this loop level. Essentially, this shows us that any two iteration vectors separated by a linear combination of a_1 and a_2 will access the same array element.

5.2. Redundant Basis Vectors

In the previous step, a basis $(a_1, a_2, \ldots, a_{n-q})$ for ker(F) was obtained. However, some vectors of the basis may be *redundant*. A *redundant* basis vector here means that a basis vector a_j of $ker(F)$ does not contribute to the solutions inside iteration-space polytope, *i.e.* if $x = t_1a_1 + t_2a_2 + \ldots + t_ja_j + \ldots + t_ka_k$, where $t_1, t_2, \ldots t_k$ are integers and also $x \in K$ then it follows that $t_j = 0$. If this were true, it would mean that the corresponding basis vector a_j could be removed as it never plays a role in data reuse. This is an important observation, as it would reduce the constraints on the derived mapping $g(x)$ imposed by validity condition (i).

From the Hermite form, we can transform linear constraints defining the polytope K into linear constraints on the coefficients of the basis vectors. For example, from (2) and the difference polytope $K = \{(x_1, x_2, x_3)| -1 \leq x_1, x_2 \leq 1, -5 \leq x_3 \leq 5\}$, we obtain the transformed inequalities $-1 \leq t_2 \leq 1, -5 \leq -t_1 + 25t_2 \leq 5$, and $-5 \leq t_1 + 25t_2 \leq 5$.

Fourier-Motzkin elimination combined with floor and ceiling functions can be used to identify redundant basis by identifying upper and lower bounds on each coefficient in turn. Proceeding with the example code, if we project the polytope K onto the basis vectors a_1 and a_2 followed by the floor and ceiling functions, then we can get the result $-3 \leq t_1 \leq 3$ and $\lceil -0.12 \rceil \leq t_2 \leq \lfloor 0.12 \rfloor$. Thus, $t_2 = 0$ and the corresponding basis vector can be removed.

Graphically, it can be seen from Fig. 3 (b) that the projection of the polytope onto the t_2 axis does not include any non-zero integral points. Thus, we obtain a minimal vector set $\{a_1 = (0, 1, -1)^T\}$ for P. Essentially, this tells us that the basis vector a_2 was superfluous, and iterations (i, j, k) and $(i, j+1, k-1)$ access the same memory location.

5.3. Subspace orthogonal to P

We have now reached the problem of how to build the modular mapping $g(x) = Gx \ mod \ s$ satisfying our conditions. In [4], it is demonstrated that to build a modular mapping one needs only derive the kernel of this mapping, which uniquely defines the mapping itself. The kernel of this mapping can be expressed as a lattice Λ, an integral linear combination of n n-dimensional basis vectors. Therefore, the problem is equivalent to finding n basis vectors in $\mathbb{Z}^n$. Validity condition (i) can be satisfied simply by including the basis already found for P within this set of vectors. Assuming fewer than n basis vectors for P were found (in our example, one basis vector for P was found and $n = 3$), Λ can be constructed by augmenting these vectors with further basis vectors, whose sole purpose is to reduce the size of the memory required, taking advantage of validity condition (ii) to reduce the required on-chip memory size.

In the proposed method, these further basis vectors are chosen from a vector subspace $P^{\perp}$ in $\mathbb{Z}^n$ orthogonal to the basis found for P, and outside the projection of the polytope K onto this orthogonal subspace. As a result we can be assured that the mapping is *valid*, as (i) the basis vectors for P are elements of the kernel of $g(x)$, thus *data reuse is preserved*, and (ii) no non-zero linear integral combination of basis vectors for P and further basis vectors can sum to zero, thus *the same memory element is never used for two different elements of the original array*.

The vector set $\{ a_1, a_2, \ldots, a_m\}$, the basis of P, has already been formed from the previous step. Now we want to choose the further $n - m$ vectors $a_{m+1}, a_{m+2}, \ldots, a_n$ from the vector subspace $P^{\perp}$. The first step is to find the subspace $P^{\perp}$ orthogonal to the vector set for P. Let Q be the $m \times n$ matrix with a_i as its ith row vector. Then a basis for $P^{\perp}$ is given by again applying the Hermite normal form algorithm, this time to the system of linear equations $Qx = 0$, representing the orthogonality condition.

For the example code, the basis for P is $\{a_1 = (0, 1, -1)^T\}$. Thus the matrix built according to our approach is $Q = (0, 1, -1)$, and the basis for $P^{\perp}$ is $((1, 0, 0)^T, (0, 1, 1)^T)$. This plane is illustrated graphically in Fig. 3 (a). Intuitively, if we restrict our selection of additional basis vectors to only come from this plane, then we can be assured not to violate validity condition (ii).

5.4. Project the original polytope K onto subspace $P^{\perp}$

In this step, the goal is to project the original polytope K onto $P^{\perp}$ to construct a polytope K', which carries properties of the polytope K and is perpendicular to the set P. Such a polytope is required because we may then choose the $n - m$ basis vectors from the vector subspace $P^{\perp}$ and outside the polytope K', completing the construction of Λ.

For the example code, the projection of the original polytope K onto $P^{\perp}$ is shown in Fig. 5, where the integral points in K' are in black.

5.5. Construction of lattice Λ

We follow the method of [4] to choose $n - m$ basis vectors $a'_{m+1}, a'_{m+2}, \ldots, a'_n$ in the subspace $P^{\perp}$, which form a basis for a lattice which intersects the polytope K' only at the zero vector.

Intuitively, the idea is to find a dense lattice of integral points that intersects polytope K' only at the zero point.

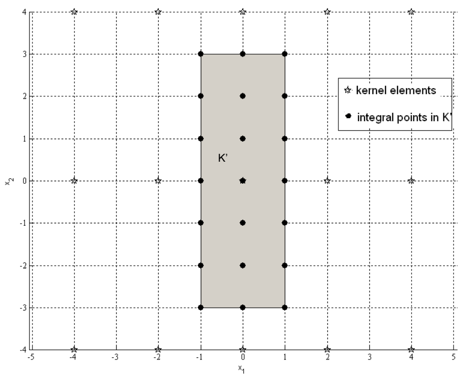


Figure 5. K'**: the projection of** K **onto** $P^{\perp}$

The restriction that it intersects only at zero corresponds to validity condition (ii), and the requirement for a *dense* lattice corresponds to a requirement for minimal on-chip memory size. Indeed, it is proved in [4] that the memory requirements will be bounded from above by the determinant of the lattice[5]. The exact algorithmic details on this step are omitted; the reader is referred to [4] for a more full discussion.

For the example code, this method results in two basis vectors $a'_2 = (0,4)^T$ and $a'_3 = (2,0)^T$ for a lattice which only intersects K' at 0. In particular, note that these two basis vectors and their combinations, shown as stars in Fig. 5 are outside K'.

These basis vectors $a'_{m+1}, a'_{m+2}, \ldots, a'_n$ are then transformed into the space of the original polytope K to get the required further $n-m$ vectors $a_{m+1}, a_{m+2}, \ldots, a_n$. As a result, we obtain a basis $(a_1, a_2, \ldots, a_n)$ the lattice Λ.

For the example code, the two basis vectors a'_2 and a'_3 are transformed into $a_2 = (0,4,4)^T$ and $a_3 = (2,0,0)$. As a result, we obtain the basis given in (3) for a lattice Λ consisting of the one reuse basis vector and two basis vectors which are perpendicular to the former.

$$a_1 = \begin{pmatrix} 0 \\ 1 \\ -1 \end{pmatrix}, \; a_2 = \begin{pmatrix} 0 \\ 4 \\ 4 \end{pmatrix}, \; a_3 = \begin{pmatrix} 2 \\ 0 \\ 0 \end{pmatrix}. \quad (3)$$

5.6. Modular Mapping Construction

After a lattice Λ is obtained, a modular mapping can be formed following Proposition 2 in [4]. Briefly, this involves forming a matrix A whose columns are the basis of the lattice and obtaining the Smith form[6] [12] of A by $S = GAU$ where G and U are unimodular matrices. This factorization provides the required modular mapping directly as $g(x) = Gx \bmod s$, where $S = \text{diag}(s)$, a square diagonal matrix with elements of s on the leading diagonal. The memory size required by this mapping is no more than the determinant of the lattice $\prod_{i=1}^{n} s_i$ [4].

As discussed, the modular mapping $Gx \bmod s$, generated using the approach we propose, is valid for the set P and the polytope K. Every x in ker($Gx \bmod s$) has a representation as an integral linear combination of vectors $a_1, a_2, \ldots, a_n$, $x = \underbrace{t_1 a_1 + \ldots + t_m a_m}_{P} + \underbrace{t_{m+1} a_{m+1} + \ldots + t_n a_n}_{P^{\perp}},$ where $t_1, t_2, \ldots, t_n$ are integers.

It therefore follows that: (i) if $f(x) = 0$ then $Gx \bmod s = 0$ because $P \subseteq$ ker($Gx \bmod s$), and (ii) any other elements in the ker($Gx \bmod s$) are outside the polytope K. Overall, the obtained modular mapping $Gx \bmod s$ is a valid mapping for the data reuse set P and the iteration difference set D.

For the example code, following this procedure results in the modular mapping with $G = \begin{pmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 1 & 1 \end{pmatrix}$ and $s = (1,2,8)^T$. Under this mapping, the required memory size is no more than $1 \times 2 \times 8 = 16$. In this example, the memory size is reduced 3.5 times. We shall show in the Section 6, that also for real benchmarks the reduction in memory size can be considerable.

It is worth noting that while it is clear that a modular mapping $Gx \ mod \ s$ uses no more than $\Pi_{i=1}^{n} s_i$ memory elements, this may not be a tight bound. In general, the 'modulo' part of the mapping may be unused, resulting in a potentially tighter bound of $\Pi_{i=1}^{n} \min\{s_i, (\max_{x \in \mathcal{IS}}(g_i x) - \min_{x \in \mathcal{IS}}(g_i x) + 1)\}$, where g_i is the ith row vector of the transformation matrix G and x is an iteration vector inside the iteration space IS. It is this bound that we shall use in the results section. For the example code, this reduces the memory requirement from 16 to 14.

5.7. Loading the Data: Inverse Mapping

The method outlined above can generate a modular mapping to replace the original array index expression in the code. However, as shown in Fig. 2 (c), the code must also initialize the reuse array by copying data from original array in external memory. Hence for each element of the reuse array, we must be able to correctly identify, with minimal control overhead, the corresponding element in the original array. As a result of the modulo operation, the mapping reuse array index to original array index may, in general, be piecewise-affine.

[5]The determinant of a lattice formed by integer linear combinations of basis vectors $a_1, \ldots, a_n$ is the determinant of the matrix with ith column equal to a_i

[6]An integral matrix A can be diagonalised by unimodular matrices G and U, resulting in the Smith form $S = GAU$ of the matrix.

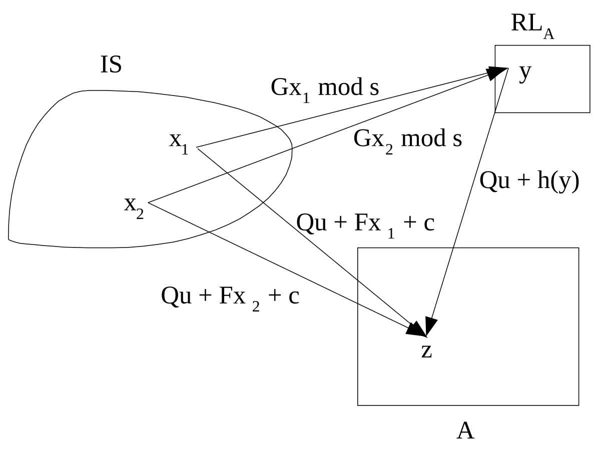
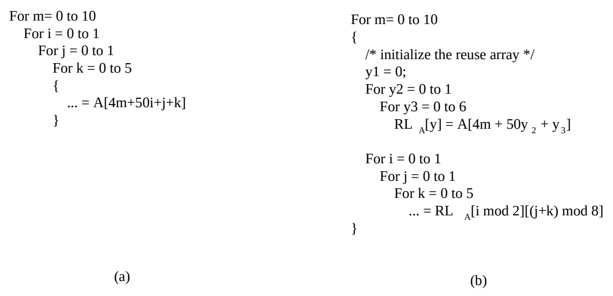


Figure 6. The relationship between the iteration space $\mathcal{IS}$, the original array A, and the on-chip reuse array RL_A. More than one element in $\mathcal{IS}$ can map to the same element of each array.



```
For m= 0 to 10
   For i = 0 to 1
      For j = 0 to 1
         For k = 0 to 5
         {
            ... = A[4m+50i+j+k]
         }
```

(a)

```
For m= 0 to 10
{
   /* initialize the reuse array */
   y1 = 0;
   For y2 = 0 to 1
      For y3 = 0 to 6
         RL_A[y] = A[4m + 50y_2 + y_3]

   For i = 0 to 1
      For j = 0 to 1
         For k = 0 to 5
            ... = RL_A[i mod 2][(j+k) mod 8]
}
```

(b)

Figure 7. Example code (continued): the indexing function of RL_A is the new modular mapping Gx mod s. (a) Original program; (b) Program with the modular mapping.

The index expression $y = Gx\ mod\ s$ derived by the proposed approach results in unimodular (and hence invertible) G, as discussed above. However, the modulo operation complicates the inverse. Re-writing the preceding expression as $y = Gx + diag(s)k$, reveals the linear nature of the problem.

For a particular reuse array index y, our aim is to establish an affine function $h(y)$ equal to the original indexing expression $Fx + c$ for any corresponding iteration vector x reading this array index.

The obtained modular mapping is not injective because some iteration vectors are mapped to the same memory location due to data reuse, but does admit a *right inverse*, *i.e.* for each $y = Gx + diag(s)k$ there is an iteration vector x corresponding to this y. We may write $x = G^{-1}y - G^{-1}diag(s)k$. The goal must therefore be to determine a suitable k such that this iteration vector lies *inside* the iteration space. Once such an iteration vector has been found, the function $h(y)$ is directly obtained as $h(y) = FG^{-1}y + c - FG^{-1}diag(s)k$. To explain this, a diagram is presented in Fig. 6. Two iteration vectors x_1, x_2 in $\mathcal{IS}$ are mapped to the same location y of the on-chip array RL_A by $Gx\ mod\ s$, and mapped to the same location z of the original array A by $E(u,x) = Qu + Fx + c$ (refer to Fig. 2) as well. The datum of these two locations is the same. y can be inversely mapped to either one of x_1 and x_2 by constructing an appropriate inverse of the modular mapping, say x_1. Then, from the iteration vector x_1, $Qu + h(y)$ where $h(y) = Fx_1 + c$ will access z of A. Finally, the datum is copied to the memory location y of the data reuse array RL_A from the location z of the original array A.

As a result, the problem can be formulated as follows. For each y in the range $0 \leq y \leq s - 1$, determine an integral vector k such that $G^{-1}y - G^{-1}diag(s)k \in \mathcal{IS}$. Since $\mathcal{IS}$ is a convex polytope, this is a *parametric integer linear program* with variable k and parameter y. Such a problem can be solved by Feautrier's PIP software [5], and always leads a guaranteed solution by construction.

For the example code, the parameter $y = (y_1, y_2, y_3)^T$ is between $(0,0,0)^T$ and $(0,1,6)^T$, as derived from the previous step of the proposed procedure. PIP provides the solution shown below:

$$h(y) = \begin{pmatrix} 0 & 50 & 1 \end{pmatrix} y. \tag{4}$$

Finally the loop structure of the example code in Fig. 7(a) is transformed to that in Fig. 7(b). This is a complete code, explicitly including data transfer to the reuse array RL_A and data access to it with a modular mapping. Thus the example is complete.

6. Experimental results

The approach described above has been implemented in MATLAB for general regularly nested loops, except for the last step, which is implemented using PIP. The inputs to our system are a polytope expressing the inner loop iteration space and an integral matrix F of the directly derived mapping for an array buffering reused data, and the outputs are a modular mapping for the array, its inverse and a loop structure for loading the reused data. For demonstration of the ability of our approach to reduce the storage space, we have applied the approach to several kernels of image processing applications, including full search motion estimation (FSME) [2], hierarchical motion estimation (HME) existing in QSDPCM [13], and the Sobel edge detection algorithm [6]. The resulting loop structures derived

Table 1. Statistics for the benchmark set.

Benchmark	# Loops	# Targeted arrays	Array size
FSME	6	2	144×176
HME	6	2	144×176
Sobel	4	2	144×176

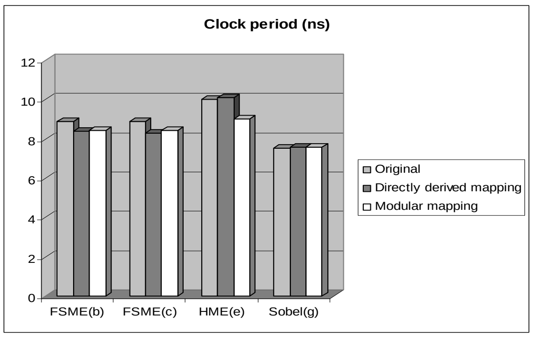
Table 2. Data reuse results. (a) original implementation without data reuse, (b) implementation with data reuse arrays for the outer-most loop, (c) implementation with data reuse arrays for the second-most outer loop.

Benchmark		(a)	(b)	(c)
FSME	# off-chip RAM accesses	4048528	40×	16×
	# on-chip BRAMs	0	3	2
HME	# off-chip RAM accesses	242608	38×	17×
	# on-chip BRAMs	0	2	2
Sobel	# off-chip RAM accesses	444744	6×	2×
	# on-chip BRAMs	0	1	1

by our algorithm have been implemented in Handel-C. The full search motion estimation is widely used and is a typical kernel for many video processing applications; its loop structure was shown in Fig. 1 (a). The QSDPCM algorithm contains a hierarchical motion estimation and our approach operates on the first level. Sobel edge detection algorithm contains a imperfectly nested loop. We assume that the benchmarks operate on 144×176 (QCIF video standard) frame size and these frames are stored in off-chip RAMs. The characteristics of the benchmarks are shown in Table 1.

Data reuse arrays have been introduced for each kernel at each level of the nested loop. Because, usually, introduction of data reuse arrays for outer loops can exploit larger data locality and require storage of a larger amount of data as well, we have introduced data reuse arrays for the outermost, second-most outer and third-most outer loops in our experiments. All algorithm versions have been implemented in Handel-C and compiled and synthesized using Celoxica DK development environment, and then placed and routed using Xilinx ISE. All results are obtained using the Celoxica RC300 platform with a Xilinx Virtex II XC2V6000 FPGA.

The first set of results concerns the reduction in off-chip memory accesses that is achievable using the scratch-pad memories. This reduction is the same, whether our approach or a directly derived address mapping [8] is used. Column (a) in Table 2 gives the absolute number of accesses to off-chip memories whereas the last two columns present



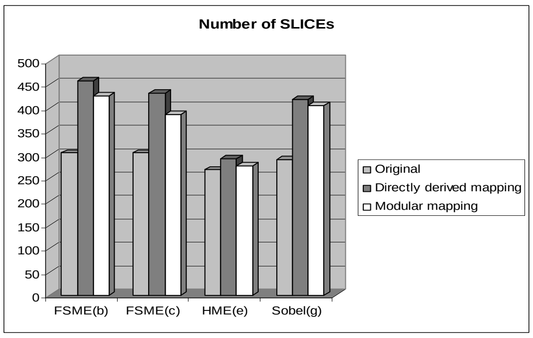


Figure 8. Clock period and number of slices. FSME(b), FSME(c), HME(e) and Sobel(g) correspond to the columns (b), (c), (e) and (g) in Table 3 respectively.

the number of times reduction over (a). We can see that introduction of data reuse arrays leads to large reductions of the number of accesses to off-chip memories up to 40 times. Under the directly derived addressing scheme, these improvements come with a relatively significant penalty in the system area. Overheads in the amount of on-chip Block RAMs (in Table 2) and slices (shown in Fig. 8) are introduced due to buffering reused data in the local memories.

In contrast, our proposed approach is capable of achieving the same reduction in off-chip communication with only a minimal on-chip memory utilization, and without worsening the system performance in clock period as can be seen in Fig. 8. We have compared the memory sizes required by the directly derived mapping, the modular mapping generated using our approach, and the actual number of distinct buffered data, obtained by fully unrolling the code. The results in Table 3 are the total storage space required by all targeted arrays in each benchmark. The memory size is measured by the number of memory elements; the width of each element remains constant across the comparison. Table 3 shows that the memory size required by the modular mapping generated using our approach is, for each case, the minimum possible, *i.e.* the same as the number of data buffered in the data reuse arrays. Compared with the mem-

Table 3. Memory requirement. (a)–(c): FSME with reuse array for the outer three loops; (d)–(e): HME with data reuse array for the outer two loops; (f)–(g) Sobel with reuse array for the outer two loops.

Benchmark	Minimum memory size (# buffered data)	Memory size (directly derived approach)	Memory size (our approach)	Improvement
(a)	2816	2816	2816	1×
(b)	160	2480	160	16×
(c)	64	1072	64	17×
(d)	712	712	712	1×
(e)	160	632	160	4×
(f)	530	530	530	1×
(g)	9	354	9	40×

ory requirement of the directly derived mapping, the modular mapping can lead to a significant reduction of more than 10 times. At the same time, we can see that in cases (a), (d) and (f) the memory requirements are the same between directly derived mapping and modular mapping, and the obtained modular mappings have the same forms as the directly derived mappings for these cases. Therefore, in practice, our approach has achieved an optimal mapping which requires the smallest storage space in all mappings for targeted statements. Due to the small frame size of QCIF under a simple mapping of the embedded memory per reuse array in our experiments, the number of Block RAMs is unchanged by this optimization, except case (b) in Table 3 that reduces the number of Block RAMs from 3 to 2. However, the reduction in storage requrement opens the opportunity of sharing Block RAMs between reuse arrays.

The compile-time computational complexity of our approach is higher than the method to directly derive mappings, and is worst-case exponential in the number of loop levels (*not* the number of memory accesses). However, it should be noted that the number of loop levels in real benchmarks is typically small (6 or fewer), and all results in this paper took less than a second to compute.

7. Conclusion

We propose an approach to automatically derive the address mappings for data reuse arrays and the inverse of these mappings so that the required memory size can be minimized. We combine a lattice-based memory reuse method into our approach, and extend the previous work in the field to incorporate data reuse characterization. We have demonstrated up to $40\times$ (average $10\times$) reduction of the memory size compared with the directly derived mapping. This reduction also results in improvement in slice count and does not worsen clock period.

The limitation of our approach is that we currently target affine array index expressions, affine loop structure, and self-reuse of each reference to an array. In the future, we will consider non-affine nested loops and group reuse between multiple references to arrays.

References

[1] F. Balasa, F. Catthoor, and H. D. Man. Background memory area estimation for multidimensional signal processing systems. *IEEE Trans. Very Large Scale Integr. Syst.*, 3(2):157–172, 1995.

[2] V. Bhaskaran and K. Konstantinides. *Image and Video Compression Standards: Algorithms and Architectures*. Kluwer Academic Publishers, Norwell, MA, USA, 1997.

[3] F. Catthoor, E. de Greef, and S. Suytack. *Custom Memory Management Methodology: Exploration of Memory Organisation for Embedded Multimedia System Design*. Kluwer Academic Publishers, Norwell, MA, USA, 1998.

[4] A. Darte, R. Schreiber, and G. Villard. Lattice-based memory allocation. *IEEE Transactions on Computers*, 54(10):1242–1257, Oct. 2005.

[5] P. Feautrier. Parametric integer programming. *RAIRO Recherche Opérationnelle*, 22(3):243–268, 1988.

[6] http://www.pages.drexel.edu/ ∼weg22/edge.html. accessed Aug. 2006.

[7] I. Issenin, E. Brockmeyer, M. Miranda, and N. Dutt. Data reuse analysis technique for software-controlled memory hierarchies. In *DATE '04: Proceedings of the conference on Design, automation and test in Europe*, pages 202–207, 2004.

[8] M. Kandemir and A. Choudhary. Compiler-directed scratch pad memory hierarchy design and management. In *DAC '02: Proceedings of the 39th conference on Design automation*, pages 628–633, NY, USA, 2002. ACM Press.

[9] M. Kvasnica, P. Grieder, and M. Baotić. Multi-Parametric Toolbox (MPT), 2004.

[10] H. J. Lee and J. A. B. Fortes. Generation of injective and reversible modular mappings. *IEEE Trans. Parallel Distrib. Syst.*, 14(1):1–12, 2003.

[11] Q. Liu, K. Masselos, and G. A. Constantinides. Data reuse exploration for FPGA based platforms applied to the full search motion estimation algorithm. In *2006 International Conference on Field Programmable Logic and Applications*, pages 389–394, 2006.

[12] A. Schrijver. *Theory of linear and integer programming*. John Wiley & Sons, Inc., New York, NY, USA, 1986.

[13] P. Strobach. QSDPCM-A new technique in scene adaptive coding. *In Proc. 4th EUSIPCO, Grenoble, France, Eselvier Publ., Amsterdam*, pages 1141–1144, Sept. 1988.

[14] I. M. Verbauwhede, C. J. Scheers, and J. M. Rabaey. Memory estimation for high level synthesis. In *DAC '94: Proceedings of the 31st annual conference on Design automation*, pages 143–148, NY, USA, 1994. ACM Press.