Reiner W. Hartenstein, Karin Schmidt: Parallelizing Compilation for a Novel Data-Parallel Architecture; J. P Gray, F. Naghdy (Eds.), PCAT-94, Parallel Computing: Technology and Practice, Wollongong, Australia, pp. 126-137, Nov. 1994

Parallelizing Compilation for a Novel Data-Parallel Architecture

Reiner W. Hartenstein, Karin Schmidt
University of Kaiserslautern, P.O.Box 3049
67653 Kaiserslautern, Germany

Abstract. The paper presents a new architectural class of high performance data-parallel machines, called Xputer. Xputer combine structural programming with traditional von Neumann control flow programming. From this combination a new programming paradigm arises which is not familiar to the usual software developer. To counteract this lack a program partitioning, restructuring, and mapping method for Xputers has been developed for the input language C. Sources are restructured and partitioned into an Xputer-suitable execution sequence providing parallelism at expression and at statement level. Data is mapped in a regular form onto the Xputer memory space to be accessible by the Xputers data sequencer hardware which provides a generic set of fast address sequences. The data operations within each part of the derived execution sequence are coded as a structural description for further synthesis towards the reconfigurable ALU which is based on field-programmable logic. Additionally, assembly code is produced in order to control program execution through the data sequencer hardware. The entire method performing the paradigm shift works without further user interaction and all steps are driven by parameters describing the actual target hardware configuration.

1. Introduction

Today we are facing increasingly complex tasks to be performed by computers. Many of these tasks are computation-intensive requiring a huge amount of high data throughput and high performance. From empirical studies ([10]) it can be concluded that the major amount in computation time is due to rather simple loop constructs. Since additionally these loop constructs are combined with indexed array data structures, ordinary von Neumann style computers are burdened with mainly addressing computations rather than actual data manipulations. First efforts to reduce addressing overhead and to introduce parallelism have been undertaken by the development of pipelined and vector supercomputers ([2], [9]). Together with the achievements in supercomputer technology parallelizing compilers have been developed, where compilation is based on data dependence analysis ([5], [11]). They have to be able to extract the parallelism from a program source and to produce executable code for different parallel target machines. Unfortunately, the hardware structures do not reflect the structure of the algorithms very well. Therefore, the compiler's task to direct the algorithm to the machine resources restricts the exploitation of inherent parallelism in the algorithm to a large extent.

Emanating from the technology of *field-programmable logic* (FPL) the new paradigm of structural programming has evolved ([6]). Instead of loading the program code as a sequence of instructions into memory (*procedural programming*), hardware structures are configured to fulfil the application needs (*structural programming*). Originally field-programmable logic has been used to accelerate the design of specific hardware. FPL technology is now available in densities that allow the configuration of complex algorithms on a small set of FPL devices within milliseconds.

A new kind of supercomputer combining the advantages of both structural programming and traditional von Neumann style procedural programming has been introduced by the architectural class of Xputers ([7]), a data-parallel machine with shared memory. One major difference to other data-parallel machines is that Xputers provide a run time customizable instruction set for data manipulations ([8]). Field-programmable logic is used to offer configurable instructions which allow a fully parallel execution in contrast to e.g. vector and other parallel computers which are mostly working in a pipelined manner. The data to be manipulated by the *reconfigurable ALU* (rALU) is addressed by a special *data sequencer hardware* containing *generic address generators*. These address generators provide a rich hardware-based repertory of patterns to access the data from the memory, consequently eliminating most addressing and control overhead. This results in high performance gains ([1]).

From the above several requirements arise for the development of a new Xputer compilation method. As input language the well-known imperative language C has been taken. To achieve the necessary Xputer fine grained parallelism at statement and expression level knowledge out of the supercompiler scene has been adapted. This fine grained parallelism enables the exploitation of the different reconfigurable ALU (rALU) subnets of the Xputer. A second major issue in Xputer program compilation is the extraction of the program's data and its mapping and distribution in a regular way over the Xputer memory space. This data arrangement together with the extracted data dependences and data accesses determine the required data sequencing and thus substantially contribute to the efficiency and performance of the program execution. The proposed compilation method is working without further user interaction and is flexible in order to be driven by the hardware constraints of the actual prototype hardware target.

Before the parallelizing compilation method is explained, the Xputer target hardware is briefly sketched by introducing the Xputer prototype *MoM 3* (*Map-oriented Machine 3*).

2. The Xputer Prototype MoM 3

Many applications out of the areas of digital signal processing, image processing, electronic design automation, and others require intensively iterative data manipulations to be performed on a large amount of data, e.g. statement blocks in nested loops. The new architectural class of Xputers ([7], [1]) with its third prototype MoM 3 (Map-oriented Machine 3) is especially designed to reduce the von Neumann bottleneck of repetitive decoding and address interpreting. This bottleneck contributes a significant amount to the run time of algorithms out of these areas (90% in image processing, 58% in DSP [1]). Although the MoM 3 may serve as stand-alone machine it is currently embedded as a general-purpose co-processor in a VMEbus based workstation (see figure 1). After setup, the MoM 3 runs independently from the host computer until the complete application is processed. Setup in this case means, that the host software has to load the application data into the MoM 3 data memory, load the GAG parameter sets, the rALU configuration code and the program for the instruction sequencer into the MoM 3 control memory and initiate execution.

The MoM 3 consists of four major parts: (1) the *data sequencer* with seven *generic address generators* (GAGs), an *instruction sequencer* (IS), and a *control memory*, (2) the *reconfigurable ALU* (rALU) with seven *scan windows*, (3) the *data memory*, and (4) the *MoMbus* for the connection of the components (see figure 1).

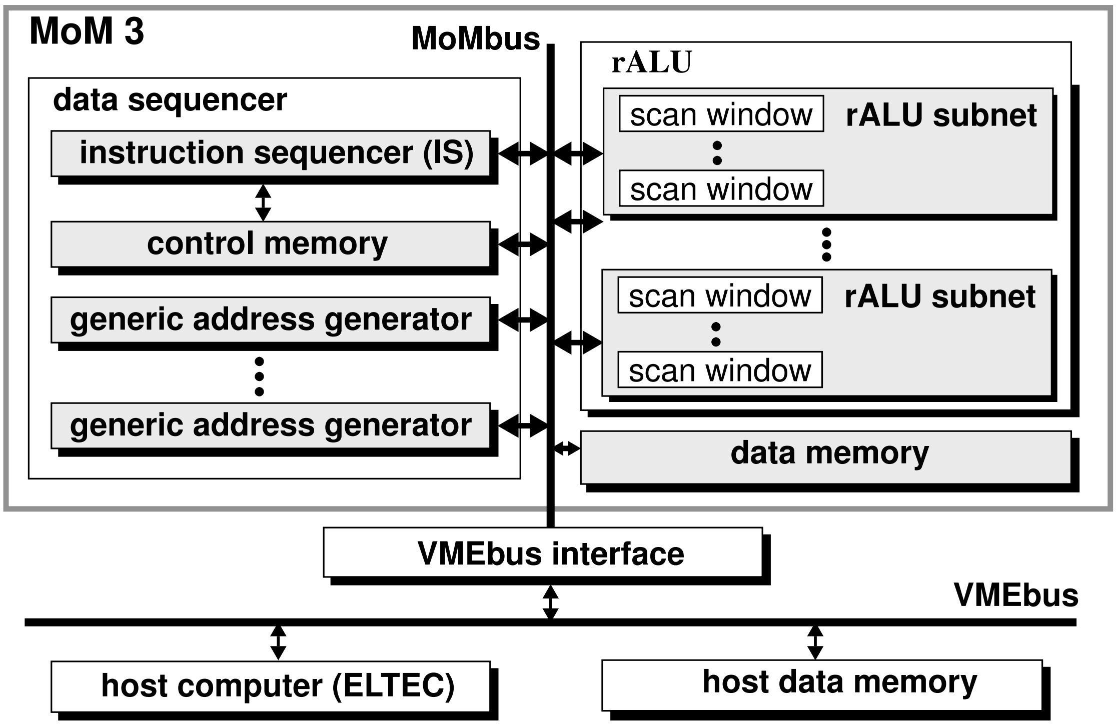


Figure 1: MoM 3 block diagram

In contrast to the one-dimensional von Neumann memory space, the Xputer's *data memory* is primarily organized to be two-dimensional by splitting the memory address into an x- and y-part like coordinates in a two-dimensional map. It holds the data of the user's programs. The data is distributed in a regular fashion over the data memory given by a mapping scheme (data map). Each *scan window* out of the rALU serves as a window to the data memory being the processor-to-memory interface of Xputers. A scan window may hold up to 128 data words from a local neighbourhood as a copy out of data memory. It efficiently supports the exploitation of parallelism within an algorithm. Scan windows are adjustable in size during run time. The *data sequencer* hardware provides accessing sequences for a controlled scan window movement over the memory space (figure 2). Thus the data sequencer represents the main control part of an Xputer. It consists of seven *generic address generators* (GAGs) operating in parallel and an instruction sequencer (IS), which reconfigures GAGs and rALU when necessary. A generic address generator is able to compute a long sequence of addresses, so-called *basic scan patterns*, for the data in the data map from a relatively small parameter set. A selection of simple scan patterns is given in figure 3.

All data manipulations are done by the reconfigurable ALU applied to the data in the scan windows. For the MoM 3 a special *reconfigurable datapath architecture* (rDPA) supporting word level has been developed ([8]) for the evaluation of any arithmetic and logic expression. Each basic cell of the rDPA serves as an operator and is configured with a fixed ALU and a microprogrammed control. During the compilation the data operations are coded as structural description for further synthesis towards the rDPA.

3. The Compilation Method

A partitioning, restructuring, and mapping method is needed to translate a sequential C program into Xputer-specific parallel code. This paradigm switch shall be performed without

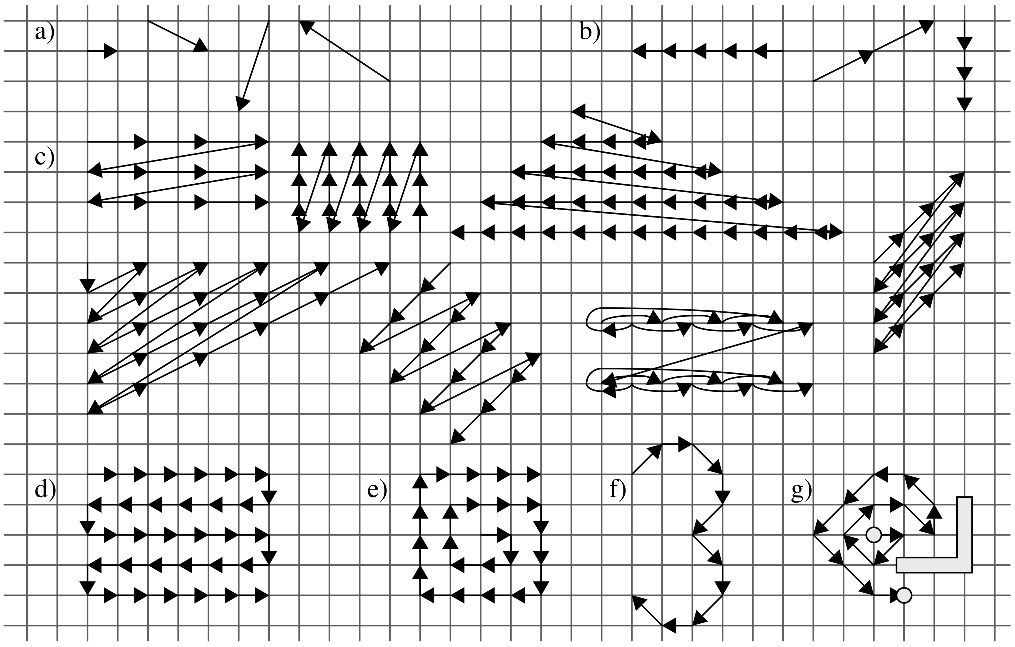


Figure 3: Basic scan patterns: a) single steps, b) linear scans, c) videoscans, d) zig-zag scan, e) spiral, f) curve following (data dependent), g) Lee Routing (data dependent)

further user interaction. The method itself deals with the fundamental problems similar to those in compiling a program for parallel execution on a multiprocessor system. These problems are: (1) Identify and extract potential parallelism, (2) partition the program into a sequence of execution units according to the granularity of the architecture and the hardware constraints, (3) compute an efficient allocation scheme for the data in the Xputer data map, and (4) generate efficient and fast code.

For Xputer compilation all these problems have to be solved during compile time. First a theory is needed for the program partitioning and restructuring (parallelization). The result of this step is the determination of a partial execution sequence (section 3.1.). Secondly the program's data has to be mapped in a regular way onto the 2-dimensionally organized Xputer data map (section 3.2.), followed by a computation of the right address accesses (data sequencing) for each variable (section 3.3.). Thus far all steps are target-hardware independent. Code generation for the MoM 3 results (1) in a *hardware image file* containing structural information for the configuration of field-programmable logic, and (2) in a *software image file* containing the parameter sets for the data sequencer hardware especially the generic address generators (section 3.4.).

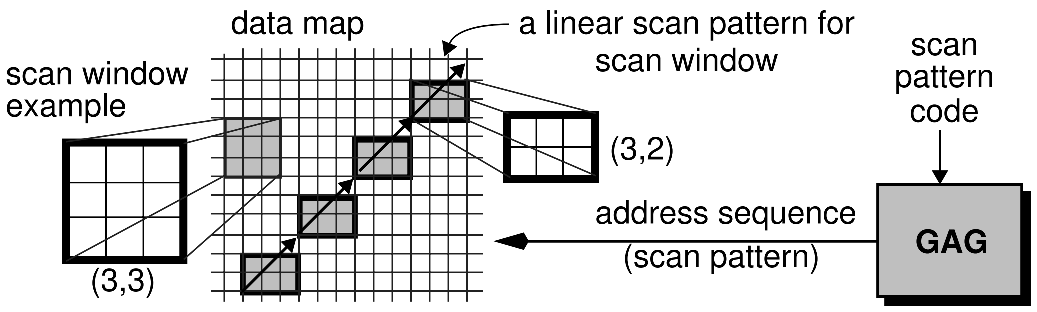


Figure 2: Different scan window sizes and an address sequence for a linear scan pattern

3.1. Determination of a Program Execution Sequence

Result of the parsing of the program is a graphical representation G = (N, E) with node set N and arc set E ([4]). The control flow of the program has to be partitioned first in order to find parallelizable subgraphs. This step is followed by a data partitioning by partial vectorization based on the level-k-dependence graph ([11]).

3.1.1. Partitioning of the Control-Flow

Given is the program graph G = (N, E). The node set N has to be partitioned, resulting in a number of subgraphs G_k, 1 k n, and an arc set E*, defining a partial execution order. The partitioning function can be given as

$$\pi : N \to N_i \,,\ 1\ i\ n, \tag{3.1}$$

with $N_i \cap N_j = \varnothing$, for i j. This function computes a number of subgraphs $G_k = (N_k^\pi, E_k^\pi)$, 1 k n, with the node set

$$N_k^\pi = \{(n \in N) | (\pi(n) \to N_k)\} \tag{3.2}$$

and the arc set

$$E_k^\pi = \{((n_i , n_j) \in E) | (\pi(n_i) \to N_k \wedge \pi(n_j) \to N_k)\} \tag{3.3}$$

The arcs of the cut set E* given by the original set E and all sets E_k^π, and defined by

$$E^* = E \cap \left(\bigcup_{k=1}^{n} E_k^\pi \right) \tag{3.4}$$

then give the partial execution order “ ”. For the structure of the subgraphs an additional criterion has to be formulated, namely that each subgraph has to be convex.The question is now, how the partitioning of a graph into convex subgraphs can be achieved. This is performed by specifying an equivalence relation on the node set N (definition 3.1) and building the corresponding equivalence classes (equation (3.6) and equation (3.7)) which represent convex subgraphs. The definition of a partial order relation for the equivalence classes (definition 3.2) then gives the execution sequence.

Definition 3.1 : *Equivalence Relation Connect*

Connect specifies an equivalence relation in the set of nodes N by

$$a \sim_{connect} b \ \leftrightarrow\ (a) = (b) \quad , \quad a, b \in N \tag{3.5}$$

Connect is reflexive, symmetric and transitive. *Connect* partitions the node set N into disjunct, non-empty equivalence classes.

$$[a]_{connect} == \{b | (b \in N \wedge a \sim_{connect} b)\} \tag{3.6}$$

The set of all equivalence classes, the *quotient of N by connect* is

$$N / connect == \{[a]_{connect} \mid (a \in N)\} \tag{3.7}$$

In each equivalence class $[a]_{connect}$ are the nodes of N which are assigned later to one Xputer execution block B_k. The node sets given by N_k^{π}, 1 k n, correspond to the associated equivalence classes of *connect* and thus represent the contents of the execution blocks. A partial order relation defines the partial execution order of the equivalence classes (definition 3.2).

Definition 3.2 : *Partial Order Relation Sequence*

A partial order relation *sequence over N / connect* is defined by

$$[a]_{connect} \quad \text{sequence} \; [b]_{connect} \leftrightarrow \text{path}\,(a, b) \qquad (3.8)$$

The partial order relation *sequence* corresponds to the set of arcs E^*(equation (3.4)). The presented method for control flow partitioning results in a coarse grained block sequence (each subgraph corresponds exactly to one block) with a partial execution order. The blocks are still target-hardware independent.

3.1.2. *Partitioning of the Data-Flow*

The goal of the next compilation step is to maximally parallelize each of the determined blocks in the sequence. This gives the basis for a good exploitation of the available hardware resources. First the level-1-dependence graph ([3], [11]) for each of the blocks is built. Since all kinds of index expressions in array variables are allowed (zero index variable, e.g. A[5]= A[2], single index variable, e.g. A[i]= A[i+2], and random index variable, e.g. A[i]= A[k+j]) a hierarchical framework of according tests is needed to determine flow, anti, or output data dependences ([11]) together with the level where the dependences exist. The level-1-dependence graph is subdivided into convex subgraphs using the same method as for the partitioning of the control-flow (section 3.1.1.). Each of the subgraphs is then maximally vectorized by applying the *Allen-Kennedy Vectorization Algorithm* ([3]). Result is a new sequence of maximally parallelized blocks containing a partial execution order. Vectorization generates a maximum degree of parallelism in a statement block of a loop nest for statements having no dependences or being not part of a recurrence or member of a cycle ([3]). But this kind of parallelization is performed independent of any resource constraints. This would surely violate the Xputer hardware constraints, since the execution cannot be realized in a pipelined mode like in a vector computer. Therefore a hardware-dependent vectorization factor is introduced. This factor is responsible to subdivide the generated vectors into smaller parts such that an optimal target hardware exploitation can be achieved.

Definition 3.3 : *Vectorization Factor VF_j*

Given is a normalized loop J with lower limit l_j and upper limit u_j and a statement S with a variable a [f (J)], f(J) is an index function. The loop has been vectorized by replacing the index function [f (J)] by $[f(l_j) : f(u_j)]$. This vector has to be partitioned by the introduction of a hardware-dependent *vectorization factor VF_j*. The factor has to be chosen such that

(1) $VF_j \in \mathbf{N}$ and

(2) $(f(u_j) - f(l_j) + 1) \bmod VF_j = 0$

and the vectorized loop has to be rewritten by:

Do J = l_j to u_j by $\mathbf{VF_j}$

S: a [f (J) **:** $\mathbf{VF_j}$] ......

The introduction of a vectorization factor VF_j has the advantage that a vectorized statement can be adapted to optimally exploit the hardware resources. The factor has to exactly divide the upper limit of a loop index function f (u_j) minus its lower limit f (l_j). Introducing a vectorization factor means that (1) the step widths of the according loops have to be adapted ("Do J = l_j to u_j by $\mathbf{VF_j}$") and (2) the index function in the variable has to be changed by the number of accessed variables at one time step ("a [f(J)**:** $\mathbf{VF_j}$]"), e.g. A [i+1, j+1:10] = C [i, j-2:10] / 2 + C [i-1, j:10]. Each vectorized loop L_j, 1 j n, may have its own vectorization factor VF_j. For all variables which are contained in the same vectorized loop the same vectorization factor has to be used. Partial vectorization together with the concept of the vectorization factor transforms each of the former coarse-grained blocks B_k, 1 k m, into a new sequence of parallelized blocks. Each of the blocks provides the special Xputer granularity and tries to optimally exploit the target hardware resources. This is the key for achieving a high performance during program execution.

3.2. *Data Mapping and Data Aligning*

The next step in compilation is to decide how the program data (variables, arrays, …) can be mapped onto the two-dimensionally organized Xputer data map $DM = \{DM_x, DM_y\}$ in a regular fashion, and how the data fields of differently mapped data variables can be aligned to a combined data field in order to use only one scan pattern.

The two-dimensional data map *DM* is in contrast to the defined arrays which have higher dimensions. This leads to the *mapping problem* resulting in the definition of a data allocation scheme. The target hardware parameters and constraints (e.g. seven GAGs are available for the MoM 3) have to be fulfilled. This leads to the *data alignment problem*. Unrolling the dimensions *I* of a variable *A* defined to be d-dimensional, d>2, and $d \in \mathbf{N}$, means to determine a function *dmap* from the index domain of the associated data object to the two-dimensional index domain of an Xputer data map DM, by

$$\text{dmap: } I_i^A \rightarrow \{DM_x, DM_y\}\text{ , with 1 } i \text{ } n. \qquad (3.9)$$

The question is what realization strategy may be chosen for *dmap*. First the dimensions in the array definition are numbered from the right to the left and are then mapped. Even numbers are mapped onto the x-coordinate (DM_x), odd numbers onto the y-coordinate (DM_y) of the Xputers data map DM. Thus the realization for the function *dmap* can be specified according to definition 3.4. Xputer dimension mapping is a kind of planarization.

Definition 3.4 : *Function dmap*

Given is an n-dimensional array A with according values V_j, one for each index I_j, 0 j n-1.

The mapping of array A can now be performed for each dimension of the data map DM = {DM_x, DM_y} by the following recursive function:

$$\text{dmap } (I_j^A) = \text{dmap } (I_{j-2}^A) * V_j^A$$

$$\text{dmap } (I_0^A) = V_0^A$$

$$\text{dmap } (I_1^A) = V_1^A\text{, with j is defined}$$

for the computation of M_x by: $j = \begin{cases} n-1, & \text{if n odd} \\ n-2, & \text{if n even} \end{cases}$

for the computation of M_y by: $j = \begin{cases} n-2, & \text{if n odd} \\ n-1, & \text{if n even} \end{cases}$

This means that the definition of an n-dimensional array A has changed from A $[V_{n-1}]$ $[V_{n-2}]\ldots$ $[V_0]$ to A´ given by

$$A' \left[\prod_{i=0}^{(N-2)/2} V_{2i+1} \right] \left[\prod_{i=0}^{(N-1)/2} V_{2i} \right].$$

A consequence of the just described dimension mapping is that each variable is treated as a separate data field. This means that one GAG has to be used for each variable. However, the Xputer prototype MoM 3 provides only a limited number of GAGs which can be used simultaneously. This produces a hard constraint for the mapping. The solution of this problem is performed in phase 2 of the mapping: the *alignment phase* of arrays (see definition 3.5). This phase combines the data fields of suitable arrays in some way for a joint application of only one scan pattern and therefore decreases the number of GAGs needed.

Definition 3.5 : *Array Alignment*

Let A and B denote arrays. *Alignment* of array A with array B means, that the mapped data space of A, called DM_A, is related with the data space of B, DM_B. This relation is formulated as:

$$\text{align: } DM_A \times DM_B \rightarrow DM_{AB}$$

The defined relation raises two main questions: (1) What are the possibilities for the realization of this relation? (2) Under what constraints is alignment of different data objects allowed for Xputers? The Xputer suitable alignment possibilities are restricted to *block alignment* and *mesh alignment*, where mesh can be further subdivided into *column-* and *row-*

cyclic.Figure 4 shows three possibilities by aligning the data fields of two variables. The first variable (´A´ in this case) always defines the reference variable.

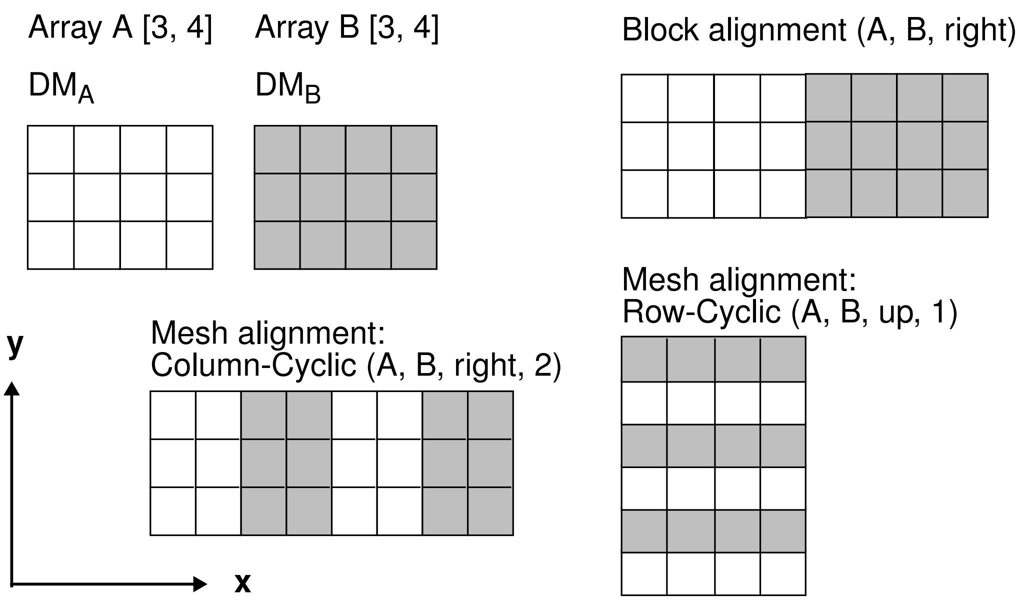


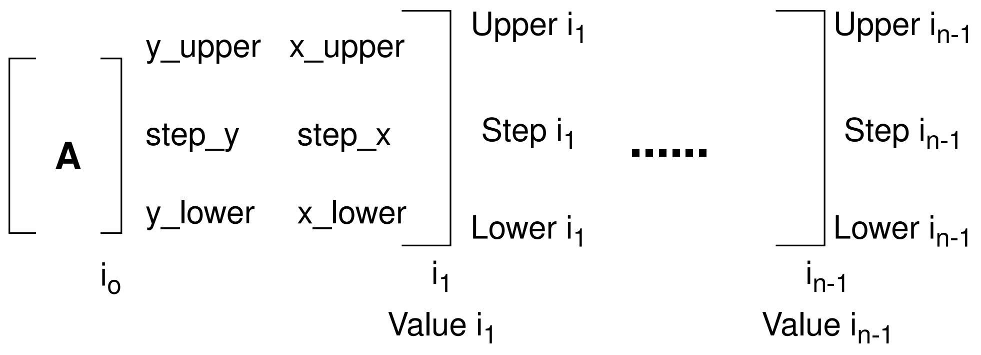
Figure 4: Examples for the alignment possibilities of the data fields of two array

Alignment is guided by a heuristic to support the selection of the 'best' choice of variables. Hence the Xputer data allocation scheme has been defined resulting in a data map description file. Bytewise initialization of the Xputer two-dimensional data memory is then performed by a special *Loader.*

3.3. Determination of the Data Sequencing

The accessing of the program data variables by their indices is needed for the generation of scan patterns (figure 3) from which the parameter sets for the data sequencer and the instruction sequencer are computed. This results in the determination of an access sequence for each variable according to their indices together with their data fields which have been mapped into a two-dimensional form. The access sequences then can be used for a computation of corresponding scan patterns and parameter sets. The computation of an access sequence is influenced by the mapping (definition 3.4), the alignment (definition 3.5), the index expressions, and the according loop limits (upper, lower, step width). Given is a m-dimensional variable *A* with index functions of random form (A [$f_{m-1}(i_{n-1},\ldots,i_1,i_0)$, …, $f_1(i_{n-1},\ldots,i_1,i_0)$, $f_0(i_{n-1},\ldots,i_1,i_0)$]. *A* is included in an n-dimensional loop nest with the innermost loop "for $i_0 = l_0$ to u_o do" and the outermost loop "Do $i_{n-1} = l_{n-1}$ to u_{n-1} do". The concept for an access sequence can then be formulated as follows:

For reasons of the two-dimensional Xputer data map the hardware of a GAG is implemented such that it generates simultaneously an address part for the y-address and an address part for the x-address. Hence the inner part of an access sequence consists of two values for the lower limit (*y_lower, x_lower*), two values for the upper limit (*y_upper, x_upper*), and two for the step widths (*step_y, step_x*). The computed values directly correspond to a fast scan pattern which is called *videoscan* ([7]). Since a loop nest for a variable *A* may be n-dimension-



$\text{Upper } i_m = \{ \text{upper } (f_k) \mid i_m \in f_k \wedge f_k \in F \}$

$\text{Lower } i_m = \{ \text{lower } (f_k) \mid i_m \in f_k \wedge f_k \in F \}$

$\text{Step } i_m = \{ \text{step } (f_k) \mid i_m \in f_k \wedge f_k \in F \}$

$\text{Value } i_m = \{ w_k \mid w_k \text{ aktueller Wert von } f_k \wedge f_k \in F \}$

$F = F \setminus \{ f_k \mid i_m \in f_k \}$ with F contains all f_k, 1 k m in the beginning

al, the basis parameter sets have to be changed after each completion of a videoscan by the values *lower* (f_k), *upper* (f_k), and *step* (f_k), 2 k n-1. This computation is handled by the instruction sequencer which configures the GAGs. All above values of an access sequence have to be computed according to definition 3.6.

Definition 3.6 : *Access Sequences and Random Index Functions*

Given is an n-dimensional loop nest with each loop according to “for $i_k := l_k$ to u_k do”, 0 k n-1, with i_{n-1} is the index variable of the outermost loop and i_0 is the index variable of the innermost loop, and the m-dimensional array variable A $[f_{m-1}(i_{n-1},\ldots,i_1,i_0),\ldots,\ f_1(i_{n-1},\ldots,i_1,i_0),\ f_0(i_{n-1},\ldots,i_1,i_0)]$. Variable A is defined by A $[V_{m-1}]\,[V_{m-2}] \ldots [V_1]\,[V_0]$. The parameters for the concept of an access sequence AS can then be computed by:

x_upper (val = u_0), **x_lower** (val = l_0):

$$\sum_{k=0}^{(m-1)/2} ((f_{2k}(i_{n-1},\ldots,i_1,\text{val}) - c) \; * \; \text{dmap}\,(I^a_{2k}) \,/\, V_{2k}) \; + \sum_{h=0}^{(m-1)/2} W_{2h}$$

$$\text{for } i_0 \in f_k\,,\ i_0 \notin f_h\,,\ \text{if } \ k \in \{0,1\} \ \text{ then } c = 0 \text{ else } c = 1.$$

step_x:

$$\sum_{k=0}^{(m-1)/2} \left(\frac{\partial}{\partial i_0} f_{2k}(i_{n-1},\ldots,i_1,i_0) \; * \; \text{dmap}\,(I^a_{2k}) \,/\, V_{2k}\right)$$

step_y:

$$\sum_{k=1}^{m/2} \left(\frac{\partial}{\partial i_0} f_{2k-1}(i_{n-1},\ldots, i_1, i_0) \;*\; \mathrm{dmap}\,(I^a_{2k-1}) \,/\, V_{2k-1}\right)$$

y_upper (val = u_0), **y_lower** (val = l_0):

$$\sum_{k=1}^{m/2} \left((f_{2k-1}(i_{n-1},\ldots, i_1, \mathrm{val}) - c) \;*\; \mathrm{dmap}\,(I^a_{2k-1}) \,/\, V_{2k-1}\right) + \sum_{h=1}^{m/2} W_{2h-1}$$

$$\text{for } i_0 \in f_k\,,\ i_0 \notin f_h\,, \text{if } k \in \{0,1\} \text{ then } c = 0 \text{ else } c = 1$$

upper (f_k) (val = u_g), **lower (f_k)** (val = l_g):

$$(f_k(i_{n-1},\ldots, \mathrm{val},\ldots, i_0) - c) \;*\; \mathrm{dmap}\,(I^a_k) \,/\, V_k$$

$$\text{for } i_g \in f_k\,, \text{if } g \in \{0,1\} \text{ then } c = 0 \text{ else } c = 1.$$

step (f_k):

$$\frac{\partial}{\partial i_g} f_k(i_{n-1},\ldots, i_g,\ldots, i_0) \;*\; \mathrm{dmap}\,(I^a_k) \,/\, V_k \quad \text{for } i_g \in f_k\,.$$

3.4. Code Generation

For each block out of the generated sequence according scan windows and access sequences have been generated, which can be now transformed into Xputer code for the MoM 3 hardware. According to the hardware structure of an Xputer (figure 1) it can be distinguished between a *hardware image file* serving as the basis for further synthesis steps and containing the structural information for the configuration of the rDPA ([8]) and a *software image file* putting the parameter sets for the data sequencing at the Xputer's disposal. The hardware image file only contains statements with arithmetical and logical expressions and no more control statements. An small example for such a file is given in the following:

```
c = d * b;
e = b / h;}
d = d mod b;
b = b - c + a * (e - 100.0);
```

The software image file includes the parameter sets for the control of the data sequencing, and the instruction sequencer, the scan windows' access strategies, and the rALU subnet activation. The first part of the file initializes the numbers of the rALU subnets and the numbers of the GAGs. The stack size is initialized, the segment sizes of the data-fields are given, and the status of the IS and according macros are defined.

For the definition of scan patterns the GAG registers (namely B0, dB, F,…) have to be described with suitable parameter sets. Each parameter set for one GAG is included in a DEFINE … ENDDEFINE block. Within this block the register values are given directly:

```
/*parameter set for a GAG*/
LABEL_SP1:      DEFINE
                    ResetAll = 0
                    AIRport = LABEL_1_MAG
                    :
                    B0.x    = 0
                    dB.x    = 0
                    F.x     = 3
                    dA.x    = 1
                    :
                    Y_Wait_X_EOL
                ENDDEFINE
```

For every address sequence which is generated by a GAG, a kind of program is needed, which tells what words of which scan window have to be loaded, which rALU subnet is used and which words have to be written back into memory. This is done by an address-scheme, ADDRScheme … ENDScheme.

```
LABEL_1_MAG:    ADDRESScheme
                    WaitAddress
                    Read        0,0
                    Read        1,0
                    WaitRalu_1
                    Write       0,0
                    Write       1,0
                    GOTO        LABEL_1_MAG
                ENDScheme
```

Furthermore the software image file includes instructions for the IS for the control of all DEFINE statements.

4. Conclusions

The paper has proposed a new data-parallel machine, the MoM 3, together with a parallelizing compilation method. The method combines structural and procedural programming, according to the Xputer paradigm of a data sequencer hardware and an FPGA-based reconfigurable ALU. For the reasons of data sequencing, which avoids repetitive address computation, high performance factors can be achieved ([1]). The parallelizing compilation method realizes the paradigm shift from von Neumann paradigm (imposed by the choice of the procedural language C) to the Xputer computing principles without further user interaction. This allows the programmer to use the advantages of a new machine paradigm without learning a new programming language. The method compiles such that the special Xputer fine granularity is achieved together with a optimized hardware exploitation of the available resources.

This fact guarantees high acceleration gains and thus offers the necessary performance for the execution of a large class of computation-intensive algorithms.

5. References

[1] A. Ast, R.W. Hartenstein, H. Reinig, K. Schmidt, M. Weber: “A General Purpose Xputer Architecture Derived from DSP and Image Processing”; in M.A. Bayoumi (ed.): “VLSI Design Methodologies for Digital Signal Processing Architectures”; Kluwer Academic Publishers, pp. 365-394, 1994.

[2] D.I. Moldovan: “Parallel Processing - From Applications to Systems”; Morgan Kaufmann Publishers, 1993.

[3] J.R. Allen, K. Kennedy: “Automatic Translation of FORTRAN Programs to Vector Form”; ACM Transactions on Programming Languages and Systems, Vol. 9, No. 4, pp. 491-542, October 1987.

[4] A.V. Aho, R.Sethi, J.D. Ullman: “Compilers - Principles, Techniques and Tools”; Addison-Wesley Publishing Company, 1986.

[5] U. Banerjee: “Dependence Analysis for Supercomputing”; Kluwer, 1988.

[6] S.D. Brown, R.J. Francis, J. Rose, Z.G. Vranesic: “Field-Programmable Gate Arrays”; Kluwer Academic Publishers, 1992.

[7] R.W. Hartenstein, A.G. Hirschbiel, K. Schmidt, M. Weber: “A Novel Paradigm of Parallel Computation and its Use to Implement Simple High-Performance-Hardware”; Future Generation Computer Systems 7, 91/92, pp. 181-198, North Holland, 1992.

[8] R.W. Hartenstein, R. Kress, H. Reinig: “A Reconfigurable Data-Driven ALU for Xputers”; IEEE Workshop on FPGAs for Custom Computing Machines, FCCM´94, Napa, CA., April 1994.

[9] K. Hwang: “Advanced Computer Architecture: Parallelism, Scalability, Programmability”; McGraw-Hill, 1993.

[10] Z. Shen, Z. Li, P.-C. Yew: “An Empirical Study of Fortran Programs for Parallelizing Compiler”; IEEE Transactions on Parallel and Distributed Systems, Vol.1, No.3, pp. 356-364, July 1990.

[11] H.P. Zima, B. Chapman: “Supercompilers for Parallel and Vector Computers”; ACM Press Frontier Series, Addison-Wesley, 1990.