21.9.1 SYS²: Mapping systolic arrays onto xputers.

We have experimented with an approach using SYS² very high level specifications (level 3 in Bild 22.11) as a source input. With this approach we have examined some program generation techniques, which transform high level specifications into an equivalent high level xputer program (MoPL-1). Since an xputer scan cache provides neighborhood communication very efficiently [56], it is most promising to adapt techniques from the area of automatic synthesis of systolic arrays [70], briefly called *systolic synthesis* ([49], [69], [72] et al.); for an introduction to systolic arrays see [67], [75], [81], or others. Systolic Synthesis makes use of nearest neighbor communication within a VLSI processor array by projecting the data dependence graph of an algorithm into time and physical processor space. Systolic synthesis can handle only *systolic algorithms* or *systolizable algorithms* (i. e. algorithms which can be converted into systolic algorithms), which are algorithms with regular data dependencies. (For a survey on systolizable algorithms see [67].) 323

Projection techniques from systolic synthesis have been adapted for parallelizing compilers for parallel computer systems. In the scene of parallel computing such techniques are called *systolizing compilation*, where the usual processes are modeled by the processing elements known from systolic synthesis, so that a concurrent implementation is derived. An xputer, however, is a monoprocessor. The problem therefore is to map the spatially distributed parallelism onto a data sequencing scheme suitable for xputers. For illustration let us use a 3 by 3 matrix multiplication example:

$$c_{ij} = \sum_{k=1}^{n} a_{ik} \cdot b_{kj} \qquad (2.28)$$

Several systolic synthesis systems, which have been implemented recently, usually accept nested loop notations as high level specifications ([49], [69], [72]). Such specifications look procedural, but the semantics is quite different: the intension is to express data dependencies, but not an order of execution (compare Bild 22.9 a and b). Expressed in SYS², the source language for the SYS³ systolic synthesis system [69], the above matrix multiplication example is specified by the following source text:

```
          for I := 1 to 3 do                                  (1)
for J := 1 to 3 do                                            (2)
   for K := 1 to 3 do                                         (3)
          C[I,J] := C[I,J] + A[I,K] * B[K,J];
```

Within the MoM-DE system we first have created a brute force solution to the problem. As a program generator we used a modified version of the systolic synthesis system SYS³, which has been available to us from a project on applications of hardware description languages. SYS³ derives a set of alternative systolic array solutions (see Bild 22.13). The xputer's compiler uses only the information which is contained in a description of the systolic array (data dependency vectors, time transformation, space transformation, index range and so on). By a heuristic search algorithm a subset of systolic arrays is selected meeting the following criteria:

- reject designs with countercurrent data streams,

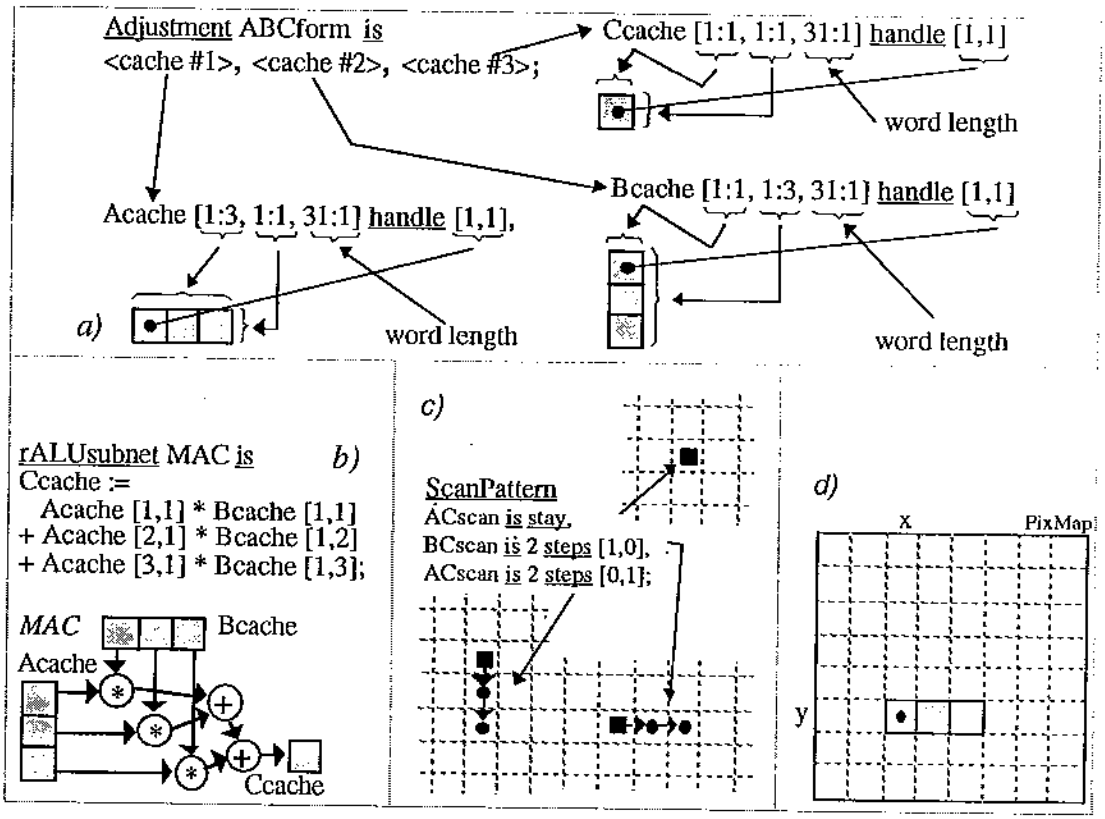

Bild 21.24: Illustrating the declaration part of the MoPL program for the multiple-cache 3-by-3 matrix multiplication: a) scan cache format adjustments ABCforms, b) compound operator MAC, three scan patterns Ascan, BCscan, and ACscan.

- reject designs with data streams flowing from more than one quadrant,
- prefer designs with the most local data streams,
- prefer designs with orthogonal streams instead of diagonal streams,
- prefer designs with all data streams flowing in one direction,

The best solution is supported, which then will be transformed into an xputer program. The scan cache size is derived from the data streams of the systolic array. The relative positions of the data queues within data streams are specified by a numbering scheme. Overlap specifies the distance by which the cache has to be

```
Array           A, B, C [1:3, 1:3, 31:0];                          (4)
                                                                   (5)
Adjustment  ABCform  is                                            (6)
               Acache [1:3, 1:1, 31:1] handle [1,1],               (7)
               Bcache [1:1, 1:3, 31:1] handle [1,1],               (8)
               Ccache [1:1, 1:1, 31:1] handle [1,1];               (9)
                                                                   (10)
rALUsubnet MAC is  Ccache = Acache [1,1] * Bcache [1,1]            (11)
                          + Acache [2,1] * Bcache [1,2]            (12)
                          + Acache [3,1] * Bcache [1,3];           (13)
                                                                   (14)
ScanPattern is  Ascan  stay,                                       (15)
                BCscan 2 steps [1,0],                              (16)
                ACscan 2 steps [0,1];                              (17)
```

Bild 21.25: MoPL declaration part for matrix multiplication example in Bild 22.14

moved. The data map is computed by the use of the space transformation function used for systolic array synthesis, by the cache size, and, by cache overlap functions [87]. 324

The program generator having been implemented at Kaiserslautern generates a MoPL-1 program (MoPL-1 is an earlier version of MoPL-3). The derivation of the data sequence needed is more difficult. For cache movements directions are selected, which are opposite to directions of data streams in the systolic array such, that data needed later are not overwritten. The time transformation used for the systolic array has to be expanded until a total time order is obtained. A Set of processor locations in a 2-D is are split into sets of processor locations on a l-D line. For each line a total order of the locations is derived. Finally, according to the "nested" orders the cache movements is arranged as a MoPL nested data sequence. Our single-cache matrix multiplication example needs 27 xputer time steps. Due to limited space we cannot go into more details.

The solution has shown, that the automatic xputer code derivation from a specification works: data map, cache adjustment, and scan pattern have been found. But this matrix multiplication solution has been rather inefficient with respect to data map size and length of the scan pattern. A solution using 3 scan caches running in parallel has been much more efficient. Next section describes a MoPL-3 program solution of this algorithm example.

21.9.2 MoPL-3: A Data-procedural Programming Language

This section introduces the essential parts of the language MoPL-3 and illustrates its semantics by means of three program text examples: the above 3-by-3 matrix multiplication, the constant geometry FFT algorithm from Bild 22.10, and the data sequencing part for the JPEG zigzag scan being part of a proposed picture data compression standard. MoPL-3 is an improved version of MoPL-2 having been implemented at Kaiserslautern as a syntax-directed editor [87].

```
begin                                                              (18)
     adjust ABCforms;                                              (19)
     apply  MAC;                                                   (20)
     moveto A[1,1], B[1,1], C[1,1];                                (21)
     fork                                                          (22)
       ACscan (Ascan), Ascan (BCscan), ACscan (BCscan);            (23)
     join                                                          (24)
end                                                                (25)
```

Bild 21.26: MoPL statement part for matrix multiplication example in Bild 22.14.

The Language MoPL-3 is an extended dialect of the programming language C. The main extension issue is the *data location* or *data state* such, that we simultaneously have two different kinds of location or state. There is the familiar von-Neumann-type *control state (current location of control)*, which e. g. is handled by *goto* statements referencing control label locations within the program text, or, by other control statements. During execution of xputer programs such a *control state* is coexisting with one or more *data location states*, what will be illustrated subsequently. (The *control flow* notation does not model the underlying xputer hardware very well, since it has been adopted from C for compatibility reasons to minimize programmer training efforts.) This also explains, why for sputers compilation techniques for control statements are fundamentally different from what the notation associates at first glance. The code to be generated from control statements is much different from that needed for von Neumann machines.

21.9.2.1 Matrix multiplication example

In addition to current control locations MoPL-3 programs also have *current data locations*, which are manipulated by `moveto` statements and scan patterns. Such a current data location is the current location of the scan cache. The statement `moveto A[1,1]`, for instance, says: move the cache to the location, where the variable `A[1,1]` is stored. A current data location does not change unless a data flow statement is encountered. I. e. after completion of a scan pattern the cache does not change its location, until another scan pattern or `moveto` statement is encountered. In case of multiple scan cache use the MoPL program has multiple current data locations. For example the statement `moveto A[1,1], B[1,1], C[1,1]` says: move physical cache no. 1 to `A[1,1]`, cache no. 2 to `B[1,1]`, and, cache no. 3 to `C[1,1]`. The following two MoPL program examples illustrate the issue of current data location. 325

Lines (4) thru (17) in Bild 22.15 show the declaration part of the matrix multiplication example. In line (4) the operand matrixes (arrays) `A` and `B`, and the result matrix `C` are declared. In line (6) thru (9) the size adjustments are declared for the physical scan caches number 1 thru 3 (also see Bild 22.14 a). The handle point preceded by the keyword `handle` indicates the particular word location within the cache, which defines *current cache location* for address generator and user. See example in Bild 22.14 d, where the current data location is `PixMap[x,y]`. In line (11) thru (13) the compound operator named `MAC` is declared (see

```
/
array              CGFFT [1:9,1:16,31:0]                              (26)
ScanPattern        InputScan  is  7 steps  [0,2];                     (27)
                   OutputScan is  7 steps  [0,1];                     (28)
                   OuterScan  is  3 steps  [2,0];                     (29)
                              •                                       (30)
                              •                                       (31)
                              •
     moveto CGFFT[1,1],[3,1],[3,9];                                   (32)
     OuterScan (  fork                                                (33)
                      InputScan, OutputScan, OutputScan;              (34)
                  join   )                                            (35)
end                                                                   (36)
```

Bild 21.27: MoPL program of constant geometry FFT scan from Bild 22.10.

Bild 22.14 b). In lines (15) thru (17) three scan patterns are declared which are named **Ascan**, **BCscan**, and **ACscan** (see Bild 22.14 c). At declaration time scan patterns are not yet assigned to a physical scan cache, nor a starting point is defined.

Lines (18) thru (36) in Bild 22.16 show the statement part of the MoPL matrix multiplication program. The **adjust** statement in line (19) assigns a predeclared format or format list (here: **ABCforms**) to physical scan caches. This adjustment remains effective until another adjustment statement is encountered. The **apply** statement in line (20) activates the predeclared compound operator named **MAC** and keeps it effective until another apply statement is activated. The **moveto** statement in line (32) is the kind of *data goto* which makes the caches no. 1 thru 3 jump into a the particular locations indicated within this statement (also see first paragraph of this section).

Line (34) shows calls to predeclared scan patterns named **ACscan** etc. (compare line (15) - (17)). These calls activate scanning actions starting from the current data locations. Note, that a call to a scan patterns is a call to a loop. The expression **ACscan(Ascan)** in line (34) indicates a call to a nested scan pattern, where the scan pattern **Ascan** is called by the scan pattern **ACscan**. This means to call a loop by a loop, i. e. to call nested loops. The **fork**/**join** brackets (22) (24) around these three scan calls show, that the three scan actions run in parallel synchronously. Due to MoPL semantics the sequence of scan calls within the fork list refers to the order of physical caches no. 1, 2, and 3.

Bild 22.18 illustrates the hierarchy of nested scan patterns of our example algorithm (rows 1 thru 3: outer loop, rows 4 - 6: inner loop) and shows the snapshots (rows 7 thru 9) of the sequence of triple cache locations created by these nested scan patterns. Whenever by SP1(SP2) a scan pattern SP1 calls a scan pattern SP2 this means, that SP1 determines nothing else than the sequence of start locations of SP2. The list of scan names within the fork/join clause refers to the (by lines (6) thru (9)) predeclared numbering scheme of physical caches **ACscan(Ascan)** is applied to scan cache no. 1, **Ascan(BCscan)** to scan cache no. 2, etc. 26 27

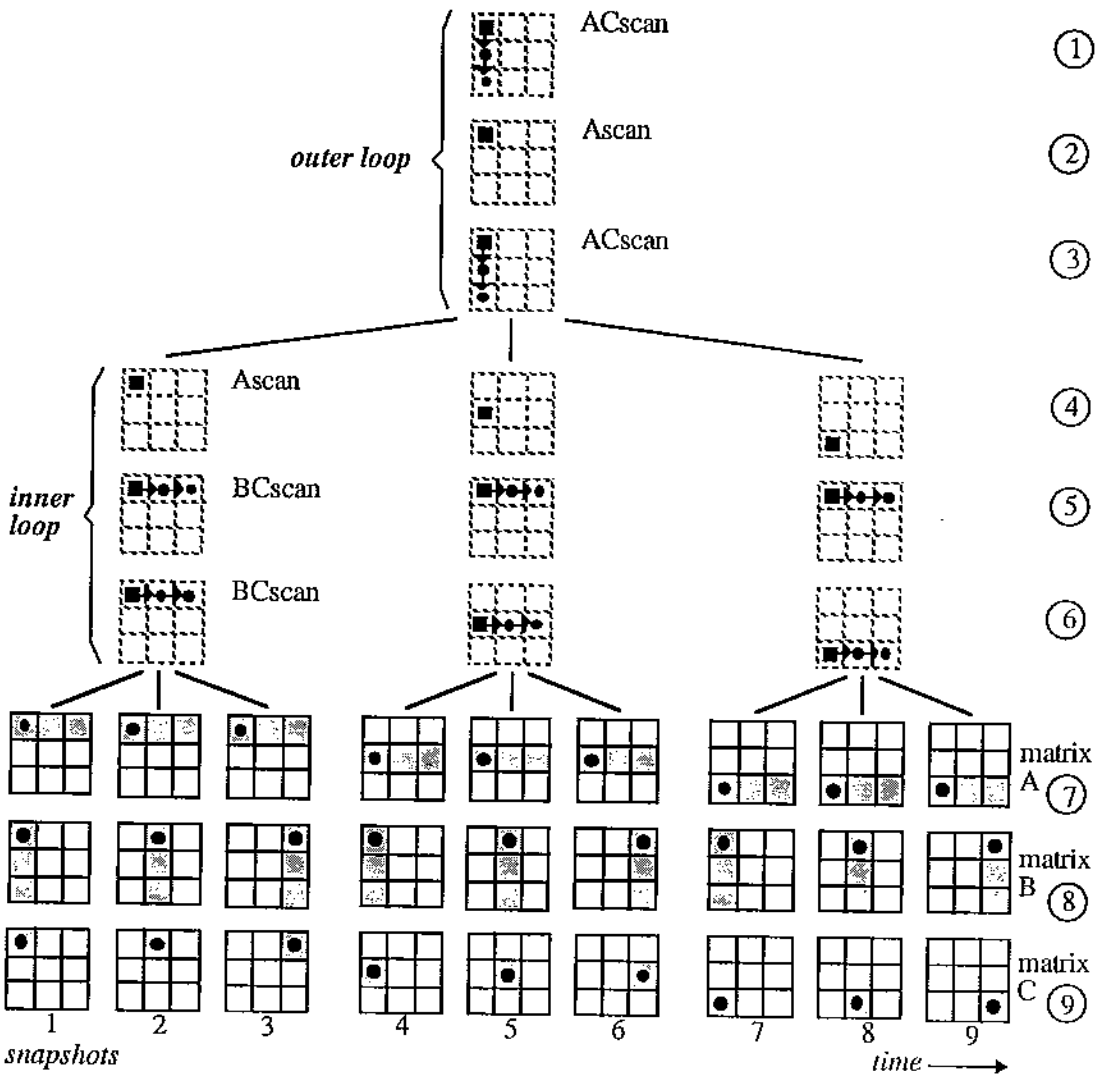

Bild 21.28: Illustration of scan pattern execution for the 3 by 3 matrix multiplication example: the hierarchy of nested scan patterns (row 1 - 3: outer loop, row 4 - 6: inner loop, row 7 - 9: snapshot sequence of scan cache locations within matrixes).

21.9.2.2 Constant geometry FFT example

Next MoPL text sample shows in lines (32) thru (35) the nested scans of the FFT algorithm example in Bild 22.10c (scan pattern declarations in line (27) thru (29).

21.9.2.3 JPEG zigzag scan example

The MoPL text from Bild 22.22 illustrates programming the JPEG zigzag scan pattern (Bild 22.20 [71]) named `JPEGzigzagScan` for scanning the array `PixMap` declared in line (37). This example uses a single 1-by-1 scan cache (adjusted as a single word buffer), which illustrates, that the performance benefit by the address generator can be obtained also for accessing long sequences of single memory locations. Lines (27) thru (29) declare four scan patterns (also see Bild 22.20), where the statements have the form:

<name_of_scan_pattern> <maximum_length_of_loop> STEPs <step_vector>.

The step vector specifies the next data location relative to the current data location (before executing a step of the scan sequence). By an escape a scan may also be terminated before <maximum_length_of_loop> is reached. E. g. see the `until` clause in line (47) indicating an escape on reaching a leftmost word within the `PixMap array` (see Bild 22.20: the first execution of `SouthWestScan` at top left corner of the array reaches only a loop length of 1). The condition `@[≤1, ]` says: escape if within current array a data location with an x subscript ≤1 is reached. The empty position behind the comma says: ignore the y subscript).

Hardware-supported Escapes . To avoid overhead for efficiency the `until` clauses are directly supported by MoM hardware features of escape execution [56] (also see Bild 22.8). To support the `until @` clauses by *off-limits escape* the address generator provides for each dimension (x, y) two comparators, an upper limit register and a lower limit register. 328

The above program covers the following strategy. The first `while` loop at lines (45) thru (50) iterates the sequence of the 4 scan calls `EastScan` thru `NorthEastScan` for the upper left triangle of the JPEG scan, from `PixMap[1,1]` to `PixMap[8,1]` (see Bild 22.20). The second `while` loop at lines (55) - (60) covers the lower right triangle from `PixMap[8,1]` to `PixMap[8,8]`. The `SouthWestScan` between both `while` loops at line (66) from `PixMap[8,1]` to `PixMap[1,8]` connects both triangular scans to obtain the total JPEG pattern.

This section has introduced the essentials of the language MoPL-3, a C extension, by means of three algorithm implementation examples. The main objective of this section has been the illustration of the language elements for data sequencing programs and the illustration of its comprehensibility and the ease of its use.

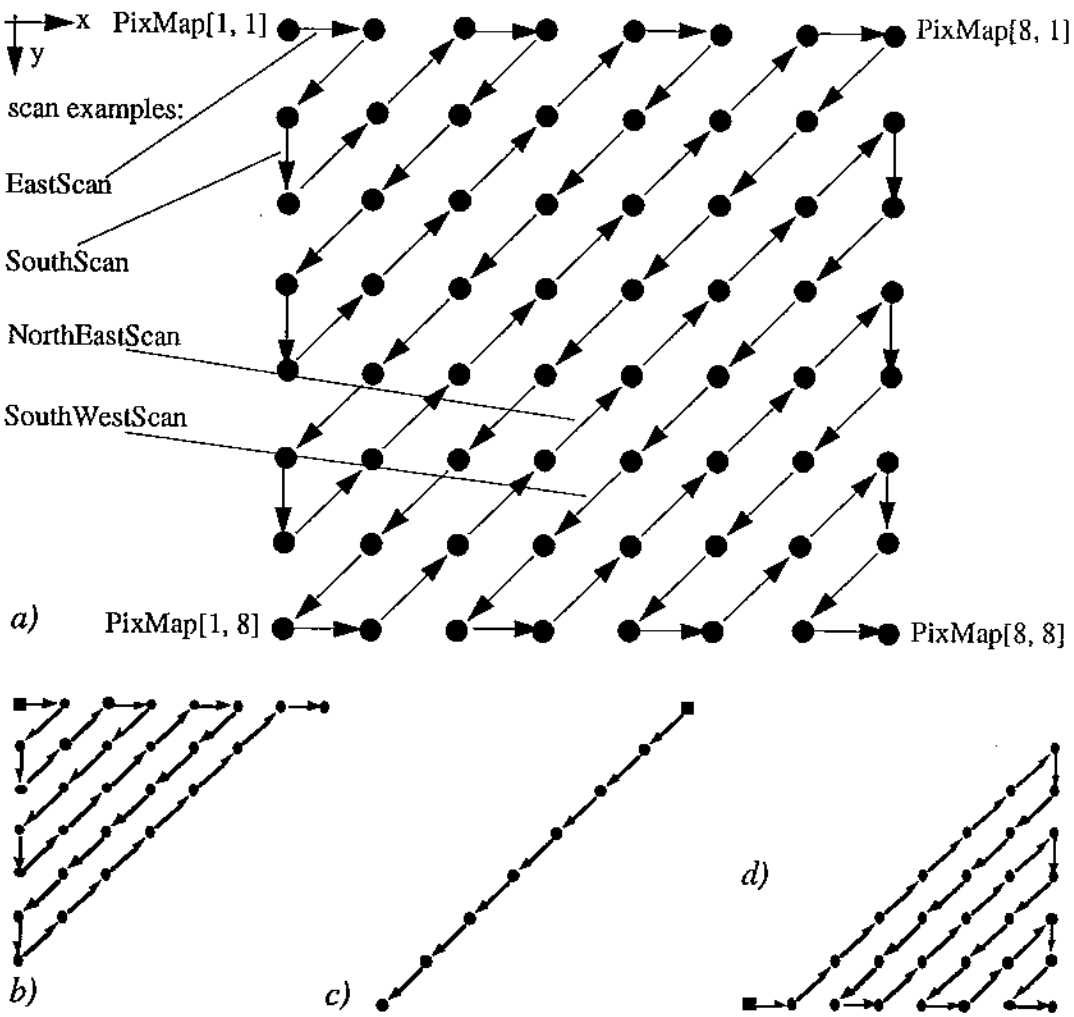

Bild 21.30: JPEG zigzag scan pattern scanning an array PixMap [1:8,1:8] (a) and its subpatterns: b) upper left triangle UpLzigzagScan, d) lower right LoRzigzagScan, c) full SouthWe-

21.10 A NEW TECHNOLOGY PLATFORM

For algorithms with regular data dependencies the Xputer paradigm is by several orders of magnitude more efficient than the von Neumann paradigm. Even for unstructured spaghetti-type sources the Xputer paradigm is at least half an order of magnitude more efficient. The high

efficiency of Xputers has several reasons:

- Their data-procedural operational principles cope much better with most kinds of overhead and bottlenecks being typical to the von Neumann machine paradigm:
    - address computation overhead
    - control flow overhead,
    - ALU (multiplexing) bottleneck
    - processor-to-memory communication bottleneck
- Xputers support some compiled fine granularity parallelism inside their reconfigurable ALU (rALU).
- *A smart register file with a smart memory interface* contributes to further reduction of memory bandwidth requirements.
- Xputers are highly compiler-friendly by supporting more efficient optimizing compilation techniques, than possible for compilers for computers.

Commercial exploitation of Xputers is now becoming feasible by the progress and commercial availability of modern field-programmable technology [50]. Seen from a global point of view the universality of the Xputer paradigm is based on a new essential technology platform, which will be explained by next paragraphs.

RAM-based hardware universality. Von Neumann machine principles are based on *sequential code*, which is laid down in a RAM and which, at run time is scanned from there by an instruction sequencer. That is why the RAM is a central technology platform for computers already for several decades. Such RAM use is a source of the elegance and the universality of the von Neumann machine paradigm. Practically all of the user-specific problem complexity is pushed into the RAM. Although the processor is hardwired an extremely high flexibility is obtained. Due to the RAM-based executions mechanism the machine code needed is highly overhead-prone [62]. This is one of the reasons of the high complexity of von Neumann machine code and the high demand of RAM space. 329

General hardware universality by a new Technology Platform. Until recently field-programmable logic and related technologies have been used mainly for prototyping of relatively simple hardware. But now some researchers have recognized, that on the basis of such a technology platform completely new computational paradigms can be developed, which cannot be obtained from RAM-based technology platforms. An example is the PAM (Programmable Active Memory) concept by Vuillemin [44]. ASIC emulators (e. g. [47],[63]), also called hardware modelers (used for simulation acceleration), are examples being available as products already for a couple of years. But all those innovative uses of field-programmable logic do not provide a procedural machine paradigm, comparable to von Neumann processors

Machine universality by programmable interconnect. Also to obtain universal *machine* paradigms, field-programmable logic is an alternative. Non-hardwired processors by using field-programmable technology platforms permit much more efficient machine paradigms. Field-programmable logic, or more precisely *interconnect-reprogrammable media* (irM), are pro-

Bild 21.31: The difference of the central technology platforms: computers and Xputers

grammable by effectively non-sequential code, quickly alterable electrically, incrementally programmable. In this context we have experimented with several implementations of the Xputer machine paradigm [56], [57], [58], using an interconnect-reprogrammable ALU (rALU). This means flexibility (universality) by adaptable processor hardware instead of RAM use. The integration density and switching speed of such media are interesting already now for a number of applications. This is just the beginning: much more can be expected in the near future from this niche of the semiconductor market. For Xputers irM (instead of the RAM) is the central technology platform (see lower row in Bild 22.21), source of simplicity, universality and elegance of hardware principles.

A new area of R&D in Programming Languages and Compilers. This new platform offers an implementation basis for a new class of programming languages and programming methods. Most of the von Neumann bottlenecks can be avoided by machine principles based on this alternative technology platform. Of course, also new compilation techniques are needed in such a fundamentally different target technology. For short term technology transfer reasons new cross compilation techniques are also needed to bridge the gap between both classes of computational paradigms.

Fast turn-around ASIC design. Recently field-programmable gate arrays have become available, which are compatible to particular real (mask-programmable) gate arrays. Due to code compatibility the personalization code of a field-programmable version can be easily translated into that of a real gate array (being faster and of higher integration density). Such conversions are carried out by *retargeting* software (e. g. [76]), which also provides an efficient bridge between computational paradigms and ASIC design. Compared to conventional ASIC this has the benefit, that simulation is replaced by execution being several orders of magnitude more efficient. Recently considerable attention has turned over to the topic of retargeting. This indicates that the high significance of retargeting for the future trends has been widely recognized.