Parallelization in Co-Compilation for Configurable Accelerators

A Host / Accelerator Partitioning Compilation Method

J. Becker

Microelectronics Systems Institute
Technische Universitaet Darmstadt (TUD)
D-64283 Darmstadt, Germany
Phone: +49 6151 16-4337
Fax: +49 6151 16-4936
e-mail: becker@mes.tu-darmstadt.de
http://www.microelectronic.e-technik.
tu-darmstadt.de/becker/becker.html

R. Hartenstein, M. Herz, U. Nageldinger

Computer Structures Group, Informatik
University of Kaiserslautern
D-67653 Kaiserslautern, Germany
Fax: +49 631 205 2640
Home fax: +49 7251 14823
hartenst@rhrk.uni-kl.de
http://xputers.informatik.uni-kl.de

Abstract— The paper introduces a novel co-compiler and its "vertical" parallelization method, including a general model for co-operating host/accelerator platforms and a new parallelizing compilation technique derived from it. Small examples are used for illustration. It explains the exploitation of different levels of parallelism to achieve optimized speed-ups and hardware resource utilization. Section II introduces novel vertical parallelization techniques involving parallelism exploitation at four different levels (task, loop, statement, and operation level) is explained, achieved by for configurable accelerators. Finally the results are illustrated by a simple application example. But first the paper summarizes the fundamentally new dynamically reconfigurable hardware platform underlying the co-compilation method.

I. Introduction

Tsugio Makimoto has observed cycles of changing mainstream focus in semiconductor circuit design and application [1] (fig. 1). Makimoto's model obviously assumes, that each new wave is triggered by a paradigm shift. The second wave has been triggered by shifting from hardwired to programmable microcontroller. The third wave will be triggered by shifting to using reconfigurable hardware platforms as a basis of a new computational paradigm.

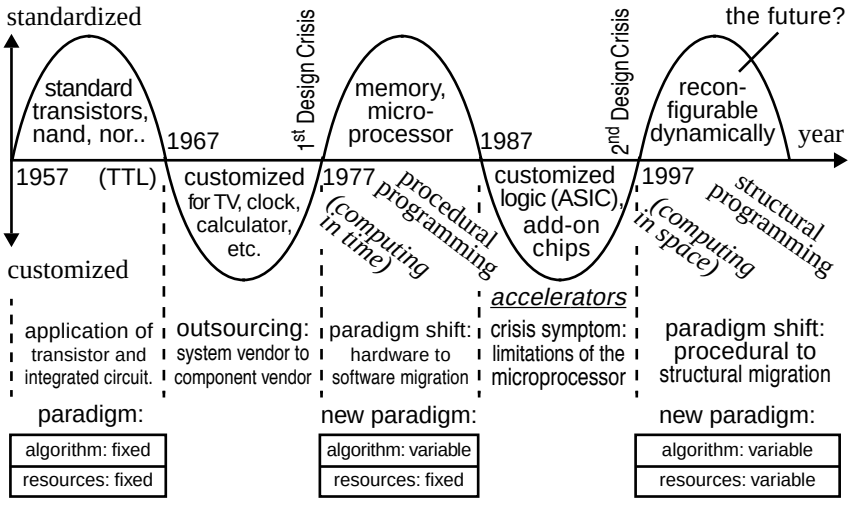
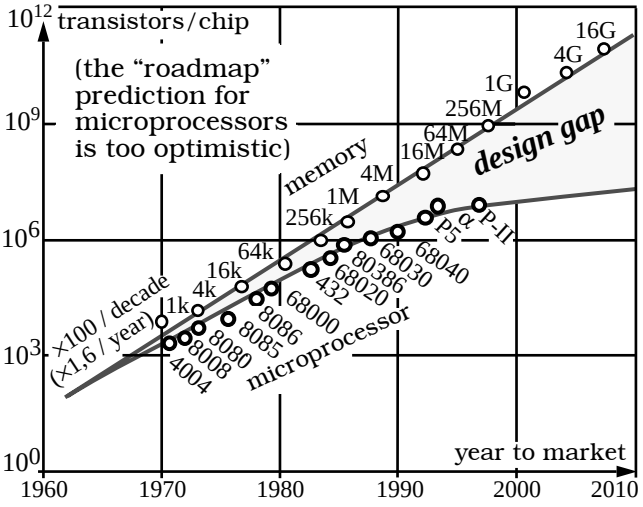


Fig. 1: Makimoto's wave: summarizing the history of paradigm shifts in semiconductor markets.

Makimoto's third wave takes into account that hardware has become soft. Emanating from *field-programmable logic* (FPL, also see [2]) and its application the awareness of the new paradigm of *structural programming* is growing. Commercially available FPGAs make use of RAM-based reconfigurability, where functions of circuit blocks and the structure of their interconnect is determined by bit patterns having been downloaded into "hidden RAM" inside the circuit. Modern FPGAs are reconfigurable within seconds or milliseconds, even partially or incrementally. Such "dynamically reconfigurable" circuits may even reconfigure themselves. An active circuit segment programs an idling other segment. So we have two programming paradigms: programming in time and in space, distinguishing two kinds of "software":

- sequential software (code downloaded to RAM)
- structural software (downloaded to hidden RAM)

But Makimoto's third wave is heavily delayed. FPGAs are available, but are mainly used for a tinkertoy approach, rather than for a new paradigm. Is it realistic to believe, that Makimoto's third wave will come? If yes, what is the reason of its delay? Although FPGA integration density has passed that of microprocessors, the evolution of dynamically reconfigurable circuits is approaching a dead end. For a change new solutions are needed for



2: The Gordon Moore curve and microprocessor curve - with design gap [4].

some fundamental issues [3]. This paper analyzes the state of the art and introduces a fundamentally new approach, which has to cope with:

- a hardware gap
- a modeling gap
- a software gap
- an education gap

A. The Hardware Gap

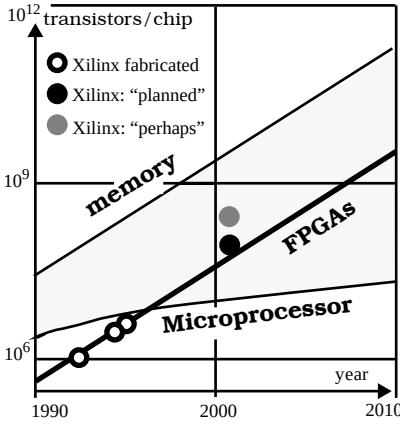
Comparing the Gordon Moore curve of integrated memory circuits versus that of microprocessors and other logic circuits (fig. 2) shows an increasing integration density gap, currently by about two orders of magnitude. We believe, that the predictions in fig. 2 [4] are more realistic than the more optimistic ones of Semicon's "road map" [5] (also [6]).

A main reason of this gap is the difference in design style [7]. The high density of memory circuits mainly relies on full custom style including wiring by abutment. Microprocessors, however, include major chip areas defined by standard cell and similar styles based on "classical" placement and routing methods. This is a main reason of the density gap, being a design gap. Another indication of increasing limitations of microprocessors is the rapidly growing usage of add-on accelerators: both boards and integrated circuits.

Both, standard cell based ASICS and FPGAs ([2], [8]), are usually highly area-inefficient, because usual placement algorithms use only flat wiring netlist statistics being much less relevant than needed for good optimization results. A much better placement strategy would be based on detailed data dependency data directly extracted from a high level application specification, like in synthesis of systolic arrays ([9],[10],[11]), where it's derived directly from a mathematical equation system or a high level program ("very high level synthesis").

Due to full custom design style FPGA integration density (fig. 3) grows very fast (at a rate as high as that of memory chips) and has already exceeded the of general purpose microprocessors [12]. But in FPGAs the reconfigurability overhead is very high (fig. 4). Figures having been published indicate 200 physical transistors needed for a logical transistor ([13],[14]) or, only 1% of chip area is available for pure application logic [15]. Routing takes up to hours of

Fig. 3: FPGA high growth rate of integration density — compared to memory and microprocessor.



computation time and uses only part of the logic elements — in some cases even only about 50%. So FPGAs would hardly be the basis of the mainstream paradigm shift to dynamically reconfigurable, such as e. g. predicted by Makimoto's wave [1] (also see analysis in [7]).

The reason of the immense FPGA area inefficiency is the need for configuration memory and the extensive use of reconfigurable routing channels, both being physical reconfigurability overhead artifacts.

B. Closing the Hardware Gap

An alternative dynamically reconfigurable platform is the KressArray [16], being much less overhead-prone and more area-efficient than FPGAs by about 3 orders of magnitude (fig. 5). (This high density may be reason to need low power design methods [18]). Also the KressArray integration density is growing a little faster than that of memories (fig. 5). The high logical area efficiency is obtained by using multiplexers inside the PEs (processing elements) instead of routing channels. Fig. 6 illustrates a 4 by 8 KressArray example. Fig. 8 illustrates the mapping (fig. b) of an application (fig. a: a system of 8 equations) onto this array.

The Kress Array is a generalization of the systolic array — the most area-efficient and throughput-efficient datapath design

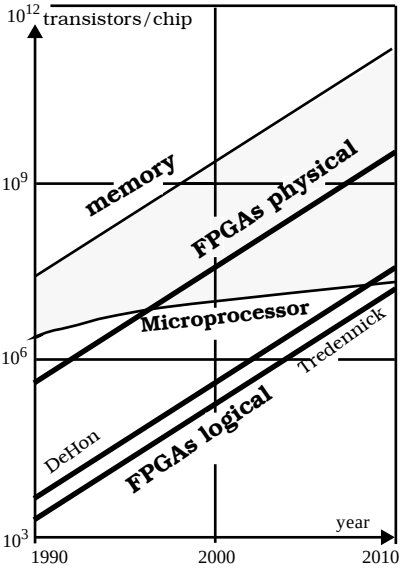


Fig. 4: The hardware gap of reconfigurability: physical versus logical integration density of FPGAs — compared to microprocessors and memory chips.

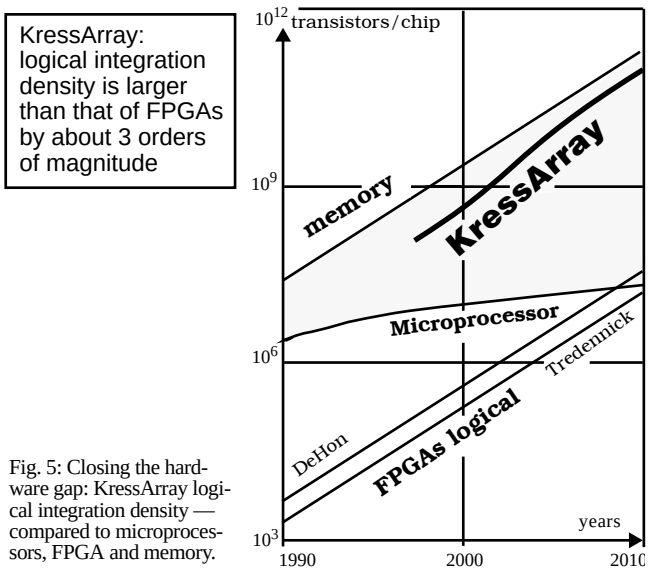


Fig. 5: Closing the hardware gap: KressArray logical integration density — compared to microprocessors, FPGA and memory.

style known, using wiring by abutment of extremely optimized full-custom essential cells, and, providing massively pipelined highly parallel solutions for highly sophisticated application algorithms like systems of equations. Systolic array methodology also includes formal design problem capture at very high level (e. g. equation systems) and elegant and concise formal synthesis methods.

The limited applicability of systolic arrays to only a small class of applications with regular data dependencies ("systolizable algorithms") is not due to its physical layout and organizational principles. The limitations are caused just by the narrow-minded mathematics-based synthesis methods (linear projections only), traditionally used for systolic arrays, where reconfigurability would not make sense because of uniformity of its highly regular results of perfectly regular interconnect: linear full length pipes only, all PEs (processing elements) having exactly the same function, etc.

By discarding the projection method and replacing it by an optimizer the design space is widened by orders of magnitude. Now reconfigurability makes sense. Results are no longer uniform. Highly flexible hardware is needed. KressArray consequences are:

applications:	any — no restrictions
interconnect [7]:	programmable at 5 levels
interconnect:	no restrictions: globally, locally & in PE individual
PE functions:	locally individual, also routing
pipeline shape:	free form: meandering, zig zag, spiral, feed back, forks, joins, ...

As an optimizer Kress uses a mapper called DPSS (data path synthesis system), being a simulated annealing optimizer [16]. Configuration code is conveyed by wormhole routing [17], which is supported by KressArray hardware. Because of the absence of routing channels the structure synthesis is carried out by a placement-only algorithm. Kress uses a simulated annealing optimizer [16]. No netlists are used: data dependency data are fully preserved to obtain optimum placement results.

The mapping problem has been mainly reduced to a placement problem. Only a small residual routing problem goes beyond nearest neighbor interconnect, which uses a few PEs also as routing elements. DPSS includes a data scheduler to organize and optimize data streams for host/array communication, being a separate algorithm carried out after placement [16]. Instead of hours known from FPGA tools DPSS needs only a few seconds of computation time. Permitting alternative solutions by multiple turn-around within minutes the KressArray tools support experimental very rapid prototyping, as well as profiling methods for known from hardware/software co-design (also see section II ff.).

In coarse granularity reconfigurable circuits like Kress-Arrays "long distance" interconnect is much cheaper and shorter than known from bit-level abutment arrays ([19] e. g. like in Algotronix FPGAs). The original Kress-Array architecture [16] has evolved to newer architectures ([7], [20]), also supporting "soft" implementations of data sequencers and other feedback datapaths, as well as of systolic arrays (this novel systolic array synthesis method is a by-product of Kress' DPSS).

C. Closing the Software Gap

The area of conventional (fine granularity like FPGAs) field-programmable logic (also see [2]) suffers also from a *software gap* [21]: only a few application development tools are available, which are difficult to use. Using reconfigurable platforms currently available commercially is mainly based on tinkertoy approaches at glue logic level. To much hardware expertise, and even also routing and placement expertise is needed for structural software implementations. We need much more powerful application development support environments, like compilers accepting application problems expressed in programming languages, like C or Java. But such structural software compilers are currently not available.

Previous section "The Hardware Gap" has also shown, that for shifting from systolic array to the much more flexible KressArray this hardware gap has been partially closed by a fundamentally different synthesis method (from mathematical methods to simulated annealing), i.e. by closing a software gap at a lower level. The achievement is the novel method of combination, since simulated annealing per se is not new.

But, although Kress' DPSS accepts C language sources it is mainly a technology mapper. It does not fully bridge the coarse granularity software gap. New software is also needed for integration into the usually embedded application environment, where automation of host/accelerator partitioning and other support is highly desired [3] (also see fig. 7).

R&D in Custom Computing Machines (CCMs: [22]), such as FCCMs (FPGA-based CCMs [23] [24]), deals with architectures and applications of dynamically reconfigurable accelerators. In some CCM architectures accelerators support a host, like e. g. a PC or workstation, so that two separate programming paths are included (also see fig. 7),

- traditional (sequential) software running on the host,
- structural software running on a reconfigurable accelerator co-processor.

The conclusion is, that the implementation of such dichotomous configware/software systems is a hardware/software

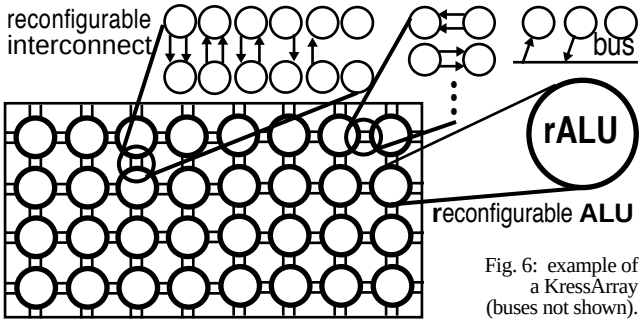


Fig. 6: example of a KressArray (buses not shown).

co-design problem, so that hardware experts are needed to "program" such platforms. To close this software gap to provide easy access by programmers a co-compiler is needed for automation of such soft-hardware/software co-design.

This new software has to cope a new class of parallelism, other than in classical parallel computing or glue logic design. It has to manage vertical and horizontal parallelization. Exploiting such parallelism with optimized code transformations in *structural programming* requires new parallelizing compilation techniques.

In traditional parallelizing compilers loop optimization techniques transform sequential loops (on *process level*) into parallelized loops (also at *process level*). This type of parallelization we call "horizontal parallelization". In contrast, parallelizing loops in *structural programming* performs a "vertical" move, one abstraction level down: sequential loops (*process level*) are transformed into parallelized loops (*data-path level*) [25]. This parallelization we call "vertical parallelization". Subsections D thru F of section II of this paper introduce a method to bridge the gap.

D. Closing the Modeling Gap

The areas of custom computing machines [26], as well as of hardware/software co-design [27] are incoherent since being torn apart by the wide variety of architectures. A general common model has been missing: the modeling gap. But for custom computing machines using coarse granularity dynamically reconfigurable datapaths like the KressArray now also a new fundamental machine paradigm is available ([28], [29]), This new paradigm might also be used for hardware/software co-design.

To implement the integration of such *soft ALUs* like the KressArray into a CCM, a deterministic data sequencing mechanism is also needed, because the traditional so-called von Neumann paradigm does not support "soft" datapaths [30], because of the tight coupling between instruction sequencer and ALU [31]. As soon as a data path is changed by structural programming, a "von Neumann" architecture falls apart and requires a new instruction sequencer.

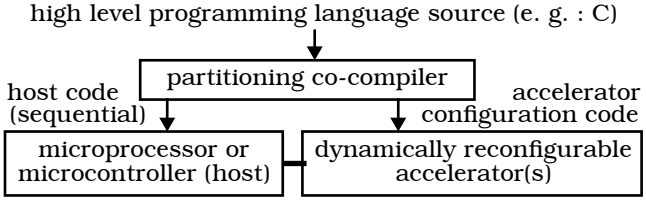


Fig. 7: Co-compilation supports host/accelerator application development.

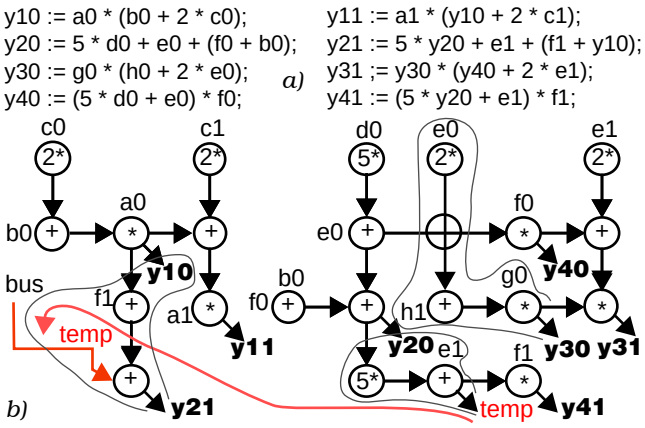


Fig. 8: A KressArray DPSS mapping example: a) application, b) result.

The solution is the use of data sequencers [32] instead of an instruction sequencer. The new computational paradigm thus obtained (published elsewhere [28], [30], also see figure 9) is the counterpart of the traditional computer paradigm, not supporting reconfigurable data paths [20]. This paradigm provides an innovative basic model for a new direction of parallel computing ([33], [34], [35]). Details and principles of the new paradigm have been published elsewhere ([28], [29], [36]). It is good backbone paradigm to close the software gap for both, coarse granularity dynamically reconfigurable platforms, as well as for the development of co-compilation methods.

E. Closing the Education Gap

Current computer science curricula do not create awareness, that hardware has become soft, nor, that hardware, structural and sequential software are alternatives to solve the same problems. Lack of awareness is blocking the paradigm shift. Intel has given courses to teach 250,000 people to enable the paradigm shift from hardwired electronics to microcontroller, what has been needed to create a market for microprocessors. A new machine paradigm, as universal as the computer [28], ready for the next paradigm shift. Section II summarizes new parallelizing compilation techniques to enter this new world of computing.

Principles and applications of dynamically reconfigurable circuits as a basis of the new paradigm of *structural programming* should be included in academic main courses to remove the mental barriers blocking the paradigm shift. But with reconfigurable platforms and related programming tools available today such a paradigm shift is not likely to happen. To create innovative expertise and awareness in many application areas new equipment of next generation reconfigurable platforms should be distributed in academia and re-

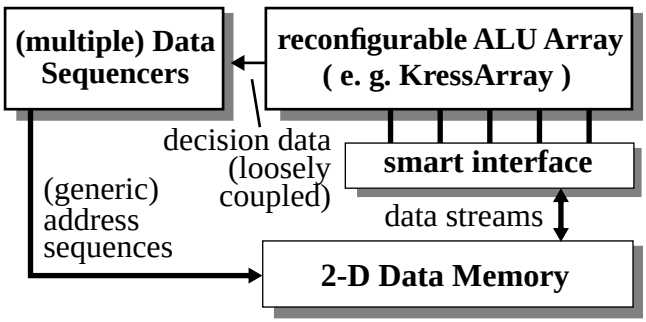


Fig. 9: Xputers: basic block diagram reflecting the basic machine principles.

search institutes, as well as new application development support software, such as e. g. introduced by this paper. Since this new area is immature, many highly significant results are expected: motivating a new generation of researchers.

F. A new Class of Custom Computing Machines

Not only in desktop or embedded systems the microprocessor's role is changing. More and more silicon area is occupied by application-specific add-on accelerator silicon [7] (for graphic, multimedia, image (de-) compression etc.), much more area than for the microprocessor itself (4th phase in fig. 1). Designing microprocessor-based systems has become a hardware/software co-design problem in general.

That's why for bridging the software gap we need more than just compilers accepting high level language sources. We need a co-compile (i. e. a partitioning compiler) providing code for both, accelerator and host (fig. 7). The co-compilation environment introduced by this paper supports accelerators based on a novel machine paradigm (the Xputer paradigm [28] [29] [36]). This is a new class of CCMs, a major step forward beyond contemporary CCMs being a tinker toy approach.

G. Reconfigurable vs. traditionally parallel

By run time to compile time migration reconfigurable platforms lead to implementations avoiding the massive run time switching overhead, known from multi-processor platforms based on traditional forms of parallelism [20] [31]. Such run time to compile time migration is a consequence of parallelism exploitation at different levels of abstraction, than known from traditional parallelism at process level. For more details see following sections.

II. Parallelizing and Partitioning Co-Compilation

Structural software being really worth such a term would require a source notation like the C language and a compiler which automatically generates structural code from it. For such a new class of accelerator hardware platforms a completely new class of (co-) compilers is needed, which generate both, sequential and structural code: partitioning compilers, which separate a source into two types of cooperating code segments ([37], [38]):

- *structural software* for the accelerator(s), and
- *sequential software* for the host.

In such an environment parallelizing compilers require two levels of partitioning:

- host/accelerator (or sequential/structural software) partitioning for optimizing performance, and
- a structural/sequential partitioning of structural software (second level) for optimizing the hardware/software trade-off of the Xputer resources.

For Xputer-based accelerators the partitioning application development framework CoDe-X (co-design for Xputers) is being implemented, based on two-level hardware/software co-design strategies [25], [39]. CoDe-X accepts X-C source programs (Xputer-C, figure 10), which represents a C dialect. CoDe-X consists of a 1st level partitioner, a GNU C compiler, and an X-C compiler. The X-C source input is partitioned in a first level into a part for execution on the host (host tasks, also

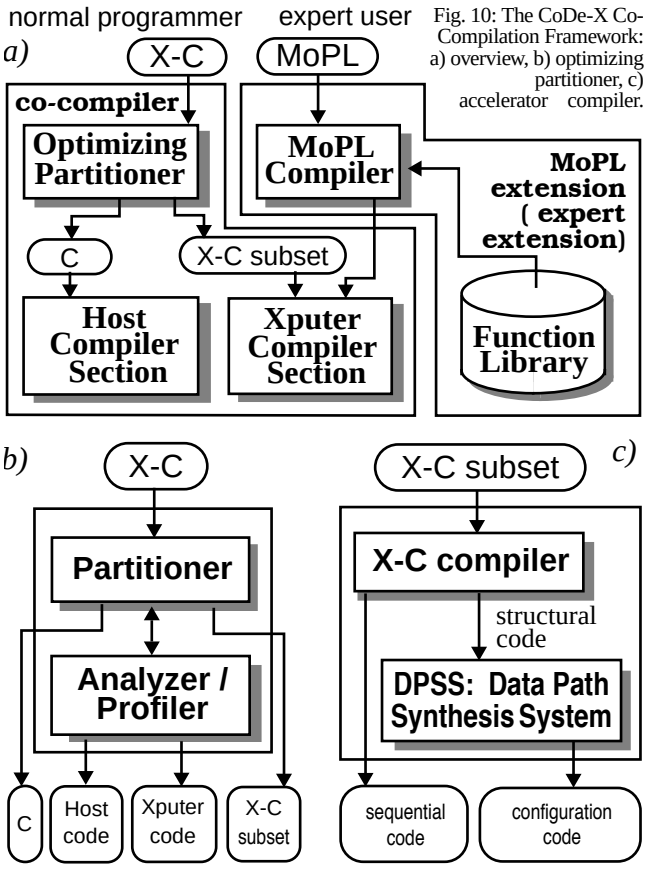


Fig. 10: The CoDe-X Co-Compilation Framework: a) overview, b) optimizing partitioner, c) accelerator compiler.

permitting dynamic structures and operating system calls) and a part for execution on the accelerator (Xputer tasks).

Program parts for accelerator execution are expressed in a C subset, which lacks dynamic structures and restricts the form of *index functions* for referencing array variables within loop bodies [25], [40]. At second level such *structural software* for configurable accelerators can be partitioned by the X-C compiler in a sequential part for the data sequencer(s), and a structural part for the rALU array.

By using C extensions within X-C experienced users may handhone their source code by including directly data-procedural MoPL code (Map-oriented Programming Language [25], [39], [41]) into the C description of an application. Also less experienced users may use generic MoPL library functions similar to C function calls to take full advantage of the high acceleration factors possible by the Xputer paradigm (see figure 10).

The MoPL language [42] provides an elegant and comprehensible method to systematically express generic data address sequences ("scan patterns") to run on data sequencers (fig. 9) [32]. With primitives like data goto, data jumps, data loops, parallel data loops, nested data loops etc., such scan patterns are the data-procedural counter part of control flow. Exercising MoPL use gives a feel of the new computational world of the paradigm of structural programming.

In subsection A the profiling-driven host/Xputer partitioning is explained first. Section E sketches the 2nd partitioning level integration into the CoDe-X framework, which is realized mainly by the X-C subset compiler [40], and section F describes the data path synthesis system (DPSS) [16], which performs fur-

ther transformation of derived structural code into loadable rALU array configuration code (fig. 10).

A. Profiling-driven Host/Accelerator Partitioning

Exploiting Task Level Parallelism: the profiling-driven first level partitioning of the dual CoDe-X partitioning process is responsible for the decision which task should be evaluated on Xputer-based accelerators and which one on the host. Generally, four kind of tasks can be determined:

- *host tasks* (containing dynamic structures,
- *Xputer tasks* (candidates for Xputer migration),
- *MoPL-code segments* included in X-C source,
- generic *Xputer library function* calls.

The *host tasks* have to be evaluated on the host, since they cannot be performed on Xputer-based accelerators. This is due to the lack of an operating system for Xputers for handling dynamic structures like pointers, recursive functions etc., which can be done more efficiently by the host. The generic *Xputer library functions* and its *MoPL-code segments* are executed in any case on the accelerator.

All other tasks are *Xputer tasks*, which are the candidates for the iterative first level partitioning step determining their final allocation based on simulated annealing ([25], [31], [39]). The granularity of *Xputer tasks* depends on the hardware parameters of the current accelerator prototype, e.g. the maximal nesting depth of for-loops to be handled by data sequencers, the number of available PEs within a KressArray, etc.

For all generated tasks corresponding host and/or Xputer performance values are determined, which are used in each iteration of the first level partitioning process for evaluating the complete application execution time, which represents the optimizing goal of this process. For details about performance evaluation in CoDe-X see [25], [43]. Since this host/accelerator partitioning methodology should be independent from an Xputer hardware prototype version, a hardware parameter file controls the partitioning. Thus, CoDe-X partitioning strategies and algorithms can be used also for future accelerator prototypes. Based thereon, the first level partitioning method of CoDe-X can be subdivided into the following major steps:

- a ***compiler front end*** performs syntax and semantic analysis of X-C input programs including also the verification of correct included *MoPL-3 code segments* as well as of *generic Xputer-library function calls* by using a MoPL-3 parser,
- a ***set of tasks*** is derived from the resulted *program graph G* (see above listed four kind of tasks), whereas the tasks' granularity depends on the given Xputer hardware parameters,
- a ***data flow analysis*** step identifies inter-task data dependencies, resulting in inter-task communications, e.g. memory re-mappings. Output of this step is a *task graph* representing all task's control and data dependence relations,
- based on the ***hyperplane concurrency theorem*** by Leslie Lamport [44], a new *vertical hyperplane theorem* has been developed [25] for performing *optimizing code transformations* to appropriate *Xputer tasks*. So potential *intra-task code parallelism* can be exploited and optimize the accelerator's hardware utilization. Correspondingly, the resulting optimized task version(s) are included in an *extended task graph*,

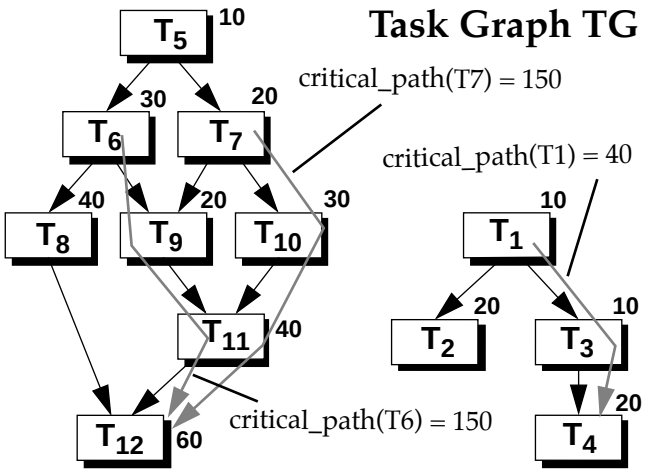


Fig. 11: Examples of critical path locations for different tasks T_i of task graph TG.

- an ***application performance analysis*** determines the performance values of each task within the *extended task graph* related to host and/or accelerator execution [25], [43],
- the ***final task allocations*** of all movable tasks will be decided in an iterative partitioning algorithm based on *simulated annealing* [25], [31]. The cost function to be minimized during this algorithm is the complete application execution time, which is estimated in each iteration by using the tasks' determined performance values and considering concurrent task executions, as well as communication-, reconfiguration, and synchronization-overhead during run time,
- finally, the ***task scheduling*** step computes the scheduling positions of all tasks within their execution queues of the host and Xputer-based accelerator modules, dependent on their data dependencies and critical path location within the task graph (see below).

Since this paper focuses on parallelism exploitation by CoDe-X, in the following the necessary *code optimization techniques*, as well as the implemented *task scheduling* step are explained. For further details see [25], [31], [39], [43].

B. Code Optimization Techniques at 1st level Partitioning

The *1st level partitioner* (see *optimizing partitioner* in figure 10) of the current CoDe-X version applies five different code transformations, dependent on the tasks' internal data flow situation, as well as on the achievable performance increase and/or the obtained accelerator hardware utilization. Therefore, to identify parallelizable index space subsets within nested loops, Lamport's *hyperplane concurrency theorem* [44] has been extended to the *vertical hyperplane theorem* [25] according to the restricted statical X-C subset of Karin Schmidt's X-C compiler [40] and its corresponding index space.

In general, Lamport's *hyperplane concurrency theorem* [44] can be applied to fully-nested loops, which satisfy the assumptions (A1) - (A5), whereas *generated variables* are a variables on left sides of assignments:

(A1) it contains no I/O statement,

(A2) it contains no transfer of control to any statement outside the loop,

(A3) it contains no subroutine or function call which can modify data,

(A4) Any occurrence in the loop body of a *generated variable* **VAR** is of the form **VAR (e^1, ... , e^r)**, where each e^i is an expression not containing any generated variable,

(A5) each occurrence of a *generated variable* **VAR** in the loop body is of the form: **VAR ($I^{j1} + m^1$, ,... , $I^{jr} + m^r$)**, where the m^i are integer constants, and j^1, ... , j^r are r distinct integers between 1 and n. Moreover, the j^i are the same for any two occurrences of VAR. Thus, if a generation A($I^2 - 1$, I^1, $I^4 + 1$) appears in the loop body, then the occurrence A ($I^2 + 1$, $I^1 + 6$, I^4) may also appear, but the occurrence A ($I^1 + 1$, $I^2 + 6$, I^4) may not.

But array variables of transformed loop nests would possibly require index functions, which cannot be expressed in the X-C subset representing *structural software*. Therefore, in addition to the *hyperplane concurrency theorem*, two more conditions (S1 and S2, see below) have to be fulfilled for transforming fully-nested X-C subset loops in equivalent X-C subset loop nests (with allowed index functions):

(S1) each *generated variable* **VAR**, which also appears on the right-hand side of assignment statements, is used within the complete body of a fully-nested X-C subset loop only in the form: **VAR(I^{jr}, . . . , I^{js}), $1 \le jr, js \le n$**,

(S1) a *generated variables* **VAR**, which appears within the body of a fully-nested X-C subset loop in the form **VAR($I^{jr} + m^{jr}$, . . . , $I^{js} + m^{js}$), $1 \le jr, js \le n$, and $\$k \in \{jr, \ldots, js\}$ with $m^k \ne 0$**, is not allowed to appear again within this loop body.

Based thereon, the *vertical hyperplane theorem* is formulized in theorem 3-1 as follows:

Theorem 0-1: Vertical Hyperplane Theorem

Assume that the loop in figure 12 (1) satisfies the assumptions (A1) - (A5), as well as (S1), (S2), and that none of the index variables I^1, ... , I^k is a missing variable. Then it can be rewritten into the form of the loop (*Par_for_all*: for all index points $\in I^k$ is the loop body parallel executable) given in figure 12 (2). Moreover, the mapping J used for the rewriting is the identity function and can be choose to be independent of the index set I.

Note, that in this *vertical hyperplane theorem* the linear mapping ϑ, used for transforming the index space of the X-C subset loop nest, is a special case: the *identity function* (see figure 12). Thus, the index functions of the resulting loop nest remain unchanged and can be still compiled in executable accelerator code, e.g. in parameters for the data sequencer(s) (performing the variable's access sequences) and configuration code for the rALU array (implementing the loop body on data path level).

Thus, Lamport's *hyperplane concurrency theorem* is suitable only for "horizontal parallelization" (process level → process level), whereas the *vertical hyperplane theorem* is the basis for a "vertical parallelization" of *structural software*, e.g. X-C subset loops (process level → data-path level). For the proof of the *vertical hyperplane theorem* and more details about this novel code parallelization method see [25].

Based thereon, the following four parallelizing transformation techniques are applied (see also [45]):

- ***strip mining***: transformation of loop nest with a large index space into several nested loops with smaller index spaces by organizing the computation in the original loop into parallel executable chunks of approximately equal size,

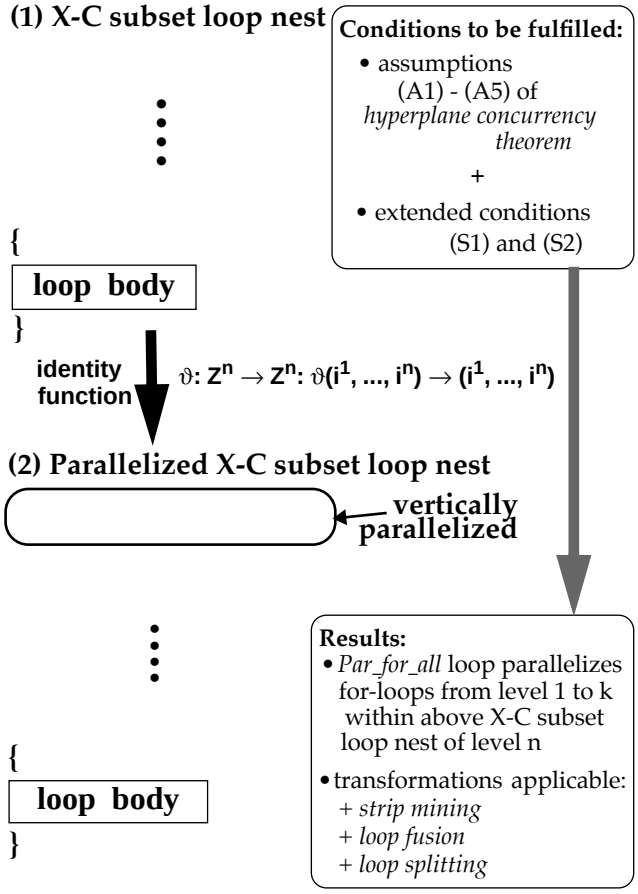


Fig. 12: Loop nest transformation à la *vertical hyperplane method*.

- ***loop fusion***: transformation of two adjacent loops into one single loop over the same index space computing the same, which reduces the number of tasks and optimizes therefore the hardware utilization,
- ***loop splitting***: reverse transformation of loop fusion, which rearranges statement instances in such a way that parts of a loop body are executed for all elements of their index space before other parts that textually follows. *Loop splitting* can partition one large task into multiple smaller tasks, each to be executed by one accelerator module without reconfiguration,
- ***loop interchanging***: transformation of a pair of loops by switching inner and outer loop, without affecting the outcome of the loop nest.*Loop interchanging* supports possible vectorization of the inner loop, and/or parallelization of the outer loop.

The fifth transformation technique implemented by CoDe-X' 1st level partitioner doesn't require a data flow analysis according to the *adapted hyperplane theorem* and is called:

- ***loop unrolling***: increasing the step width of single loop iterations and decreasing therefore the total number of iterations, by unrolling the loop body up to a certain factor. This technique is applied for increasing performance and optimizing the accelerator's hardware utilization, e.g. reducing the number of idle DPUs within a KressArray.

C. Strip mining

The transformation technique to be explained more detailed in this paper is *strip mining* (see figure 11), which exploits parallelism on task level, given within X-C input programs. For more examples, application and parallelism exploitation of the other above introduced code transformation techniques please see [25]. *Strip mining* makes it possible to compute these chunks concurrently on different Xputer-based accelerator modules.

In figure 11 the *strip mining* application of a computation-intensive loop nest within an image smoothing algorithm is shown (see also application example in section III). In our approach the block size of the chunks depends on hardware parameters describing the current accelerator prototype (e.g. the number of available accelerator modules) in order to achieve an optimized performance/area trade-off.

This technique can be used e.g. often in *image processing* applications by dividing an image in stripes of equal sizes in order to manipulate these stripes concurrently on different accelerator modules [39], whereas each strip execution represents one Xputer task.

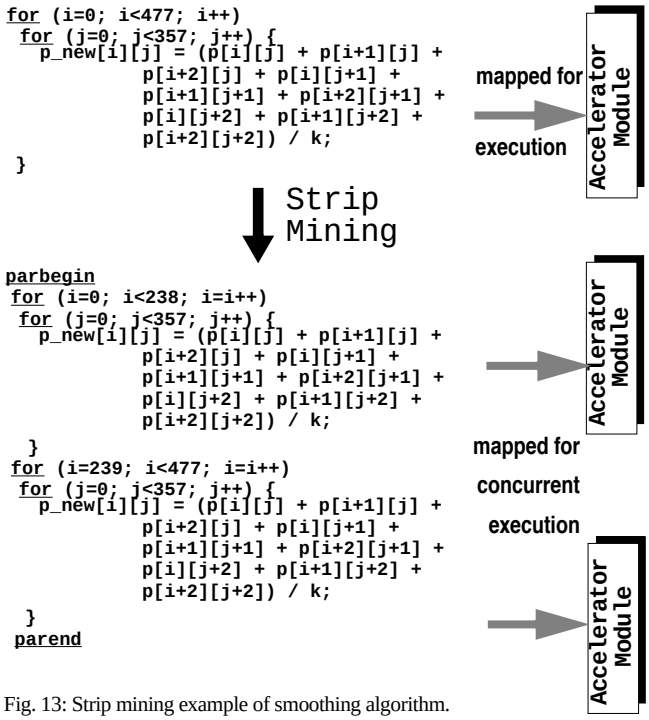


Fig. 13: Strip mining example of smoothing algorithm.

D. Scheduling and Run Time Reconfiguration(s)

The *Xputer Run-time System (XRTS)* [46] provides the software-interface from the host to Xputer-based accelerators. The main purpose of this system is controlling and synchronizing applications executed on the accelerator, but additional features like program debugging are also possible. The Xputer Run-time System can be started interactively or inside a host process and has following features:

- XRTS loads Xputer object files (sequential/structural code for data sequencer(s) / rALU Array)
- XRTS loads application data binary files with optimized distribution to the memories of different modules to minimize inter-module communication

The *run time scheduling process (RTS-process)* for performing task scheduling, communication/synchronization and reconfiguration(s) during run time activates the XRTS and is generated by CoDe-X' 1st level partitioner after the final task allocation and scheduling is determined. For details about the RTS-process and its implementation see [25].

Next, the *task scheduling* step is described first. This step determines a deadlock-free total execution order of all allocated tasks. Parallel task execution and reconfiguration are possible, according to the detected *inter-task data dependencies*. Therefore, a static *resource-constrained list-based scheduling* technique is applied by building a *task priority list* for each hardware resource, e.g. the host and its connected accelerator modules. A task is inserted into the *priority list* of its allocated hardware resource according to a two-stage *priority function*, which is explained below. Since this scheduling problem is NP-complete, appropriate heuristics have been selected for finding a good solution.

The *list-based scheduling algorithms* [47] belong to the heuristic methods. Since it is a very flexible scheduling method, it is described in detail in literature. In our case, a *priority function* for building a *priority list* is used to choose from all candidate tasks the tasks to schedule next. The list-based scheduling algorithm in our approach tries to minimize total execution time under resource constraints, and is applied as follows (see also [25]): The heuristic *priority function* for building the *priority lists* is introduced, since it is an important issue of the proposed scheduling method:

- *inter-task data dependencies* have the highest priority; for each hardware resource the list of its allocated tasks is pre-ordered according to these determined data dependencies, which results in a partial task execution order, and guarantees consistency of data dependent task executions, and,

for second priority criteria several possibilities exist [25]:

- *shortest ASAP first* or *shortest ALAP first*. Tth easiest and most straightforward way is to order the priority queues according to each task's determined ASAP- or ALAP-synchronization point in relation to the program's data flow, e.g. *shortest ASAP first* or *shortest ALAP first*,
- an related alternative is to take each task's *mobility metric* as criterion, which can be computed as the difference of the determined ASAP- and ALAP-synchronization points, e.g. *shortest mobility first*. Thus, tasks with tight data dependency constraints are scheduled first, which may handicap the execution of other tasks,
- taking each task's *critical path location* within the *task graph* as priority criterion, which is probably the most sophisticated choice in our case and implemented in the current CoDe-X version. The idea here is to schedule tasks first that have many and/or computation-intensive data dependent tasks to be executed thereafter, because this may be critical for the total application execution time.

Examples of different *critical path locations* within a *task graph TG* are illustrated in figure 11. The numbers at the upper right corner of all tasks represent some pre-computed fictitious task execution time values for host or accelerator execution, dependent on their allocation.

Based on the assumptions introduced above, the static ***resource-constrained list-based scheduling*** technique is implemented in the CoDe-X framework in building a *task priority list* for each hardware resource (host and Xputer-based accelerators) according to the following criteria:

- first criteria: data dependencies (consistency),
- second criteria: critical path location (figure 11).

The introduced *resource-constrained list-based scheduling* technique generates as output the completed *task sequence* incl. the task scheduling information, and is applied to each iteration of CoDe-X' profiling-driven *first level partitioning loop* after the *task allocation* step. The static task scheduling information is necessary for estimating the total application execution time in each iteration, which is the cost function to be minimized during the host/accelerator partitioning process. List-based scheduling algorithms are widely used in synthesis systems since they are simple to adjust on given problems with appropriate priority functions, and they differ not much from optimal ones [48].

E. Resource-driven Partitioning

To Exploit Statement Level Parallelism the X-C compiler [40] realizes the 2nd level of partitioning and translates accelerator migrated X-C subset code segments into code which can be executed on the Xputer. It divides the restricted (e.g. index functions) accelerator source code into *sequential code* for the data sequencer hardware, and *structural code* for the rALU array. The compiler performs a data and control flow analysis. The Xputer hardware structure itself provides best parallelism at statement or expression level.

Exploiting statement level parallelism [40] 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 maximal parallelized 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.

First, a theory is needed for the program partitioning and restructuring. Result of this step is the determination of a partial execution sequence with maximal parallelized execution units. Secondly the program's data has to be mapped in a regular way onto the 2-dimensionally organized Xputer Data Memory, followed by a computation of the right address accesses (data sequencing) for each variable. Code generation for Xputer-based accelerators results in structural code for the configuration of the rALU array, and more sequential code containing the parameter sets for the multiple data sequencers. For details and examples about this compilation method see [40].

F. Parallel Data Path Synthesis

After the sequential/structural partitioning step, the structural code derived from the X-C subset compiler has to be mapped onto the KressArray device. To do this, the DPSS [16] is used. The input language of the DPSS is ALE-X (arithmetic & logic expressions for Xputer). It can be edited manually, or it can be generated from the X-C subset compiler in the Xputer software environment [40].

The main steps in the process from the ALE-X input to the rALU array configuration code are the placement and routing of the operators and the I/O scheduling. The placement and routing of the operators (e.g. operations within a loop body) is performed by a simulated annealing controlled optimizing algorithm, which tries to minimize global data accesses within the rALU array. The I/O scheduling of corresponding operands realizes a kind of data scheduling with critical path data scheduled first. For details about these steps see [16], [49].

The kind of parallelism to be exploited by DPSS application mapping onto the rALU array is parallelism by pipelining at loop and operation level. The DPSS applies *loop folding* to inner loops, which performs the pipelining of loop iterations (loop level parallelism). If the DPSS receives vectorizable code from the 2nd partitioning level (see [40]), the corresponding statements can be executed in parallel (statement level parallelism). Additionally, the DPSS can pipeline vectorizable statements within the rALU array in executing them in a PE-pipeline (operation level parallelism), if not enough PEs are available for executing them in parallel.

Once the scheduling is done, the KressArray code is generated. The configuration code is generated from the placement information of the PEs and a library with the code for the operators. The sequence(s) of data words inside the scan window(s) is determined by the I/O scheduling. For more details and examples on parallelizing datapath synthesis with DPSS see [16].

III. ILLUSTRATING PARALLELIZING CO-COMPILATION

To illustrate the parallelism exploitation of the introduced compilation techniques by this paper the already used image smoothing algorithm is summarized in this section. Although being of bad quality the smoothing filter algorithm used here has been selected for its simplicity. A smoothing effect is shown in figure 14. Given an N x N image $p(x,y)$, the procedure is to generate a smoothed image $p\_new(x,y)$, whose gray level at each point (x,y) is obtained by averaging the gray-level values of the pixels of p contained in a predefined neighborhood (kernel) of (x,y). In other words, the smoothed image is obtained by using equation eq. 1:

$$p\_new(x, y) = \frac{1}{M} \sum_{(x, y) \in S} p(x, y) \qquad \text{(eq. 1)}$$

for x,y = 0,1, ..., N-1. S is the set of coordinates of points in the neighborhood of (but not including) the point (x,y), and M is a pre-computed normalization factor.

The tasks containing I/O routines for reading input parameters and routines for plotting the image are executed by the

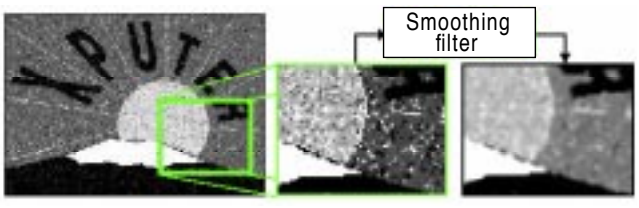


Fig. 14: Illustration of a smoothing filter operation.

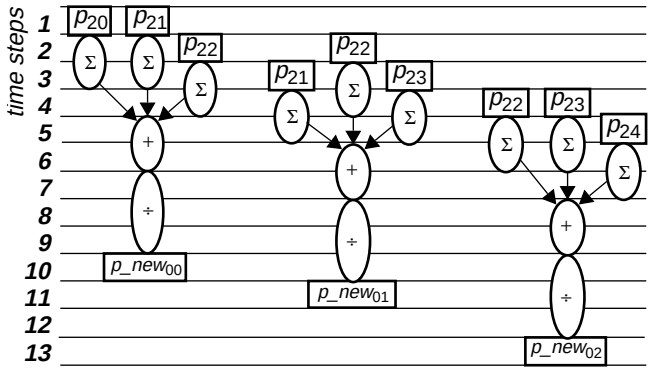


Fig. 15: Parallel execution of vectorized statements of smoothing-filter example.

host, because they cannot be executed on the Xputer. The remaining tasks are potential candidates for mapping onto the accelerator. For more details about partitioning of such image processing applications and their complete task structure please see [25]. One computation-intensive task of these potential migration candidates has been taken in section A as example task for applying the introduced compilation techniques for configurable accelerators.

Parallelism at *task level* was exploited in applying the *vertical hyperplane theorem*, and based thereon the code transformation *strip mining*, resulting in several tasks to be executed concurrently on different accelerator modules (see section A). To the inner loop of each of these tasks vectorization can be applied (see [40]), resulting in parallelism at *statement level* (if enough hardware resources, e.g. PEs within rALU array are available), or parallelism by *pipelining at operation level* (if not enough PEs are available for statement parallelization [16]). Additionally, *loop folding* (see section F) can be applied for achieving parallelism by pipelining at *loop level*.

The DPSS mapping of one vectorized task (vectorized statement of inner loop, see figure 11) incl. the time scheduling of corresponding KressArray operators is shown in figure 15. Hereby, the Σ–operator stands for summing up 3 subsequent pixel values p_{ij}. The derived structure takes the values of the p_{2j} pixel column as input to the sum operators and emits the new value for p_new_{00} from the division operator. The values of the pixels p_{0j} and p_{1j} need not to be read, as they are stored inside the sum operators from the previous steps.

The next step of the DPSS is the data scheduling of necessary operands. The sequencing graph in figure 15 contains also this scheduling information, whereas 2 operands p_{ij} can be brought over 2 parallel busses to PEs of the rALU array. For demonstration purposes, we assume a simplified timing behavior for our filter example, where an I/O operation has a delay of one time step, an addition or sum operator two time steps and a division by the smoothing constant three time steps. For more details and examples of DPSS mappings see [16].

IV. Conclusions

The KressArray, a novel dynamically reconfigurable technology platform has been briefly recalled. It has been recalled, that the KressArray is the generalization of the systolic array, but resulting in a general purpose platform. It has been shown, that for about the next ten years the integration density of KressArrays grows faster than that of memory. Long term growth will equal that of memory. The logical area efficiency of the KressArray goes by at least three orders of magnitude beyond that of FPGAs. It has been shown, that FPGAs obviously do not have the potential for a break-through.

Microprocessor and -controller applications often feature by far more silicon area for add-on accelerators than for the host, resulting in a design cost explosion. KressArray-based solutions are a highly promising alternative to accelerator implementation, also for compilation/downloading instead of design, and, by upgrading thru downloading instead of re-design.

A novel parallelizing co-compiler for coarse grain dynamically reconfigurable computing machines has been outlined — the first complete co-compilation method, bridging the software gap for such target architectures by accepting C language source programs, and generating both, *sequential software* for the host, as well as *structural software* to (re-) configure structurally programmable accelerator hardware.

It has been illustrated, how this compilation framework CoDe-X is the implementation of a two-level partitioning method, and, how the *structural software* segments are further optimized by the second level sequential/structural software partitioner, and corresponding loadable structural code, data schedules and storage schemes are generated.

The paper has contrasted the multiple parallelism levels exploitation by the novel “vertical parallelization” technique to traditional “horizontal” parallelization. This new application development approach, featuring task-, loop-, statement-, and operation-level parallelism, has been illustrated by a computation-intensive but simple example. The general model for the underlying technology platform and its “high level technology mapping” have been introduced briefly, which provide instruction level parallelism, being drastically more area-efficient than using FPGAs and similar platforms.

V. Literature

[1] D. Manners, T. Makimoto: Living with the Chip; Chapman & Hall, 1995

[2] annual int’l Workshops on Field-programmable Logic and Applications (FPL); Lecture Notes on Computer Science, Springer Verlag

[3] W. H. Mangione-Smith, B. Hutchings, D. Andrews, A. DeHon, C. Ebeling, R. W. Hartenstein, O. Mencer, J. Morris, K. Palem, V. Prasanna, H. Spaanenburg: Current Issues in Configurable Computing Research; (to appear) IEEE Computer

[4] R. Hartenstein: Wozu noch Mikrochips?; ITpress Verlag, 1994

[5] R. Weiss: Going for the gold - ASIC's go mainstream; Computer Design, Sept. 1996, pp. 79-80.

[6] R. Weiss: Coping with success, thanks to choices and tools; Computer Design, Sept. 1996, pp. 65-66.

[7] R. Hartenstein (invited paper): The Microprocessor is no longer General Purpose: why Future Reconfigurable Platforms will win; IEEE International Symposium on Innovative Systems in Silicon, Oct. 1997 (ISIS’97), Austin, Texas, U.S.A.

[8] annual IEEE Workshops on FPGAs (FPGA)

[9] Y. S. Kung: VLSI Array Processors; Prentice-Hall 1988

[10] N. Petkov: Systolische Algorithmen und Arrays; Akademie-Verlag 1989

[11] N. Petkov: Systolic Parallel Processing (Advances in Parallel Computing, Vol 5); North Holland 1993

[12] N. N.: Programmable Logic Breakthrough ‘95; Tech. Conf. and Seminar Series, Xilinx, San José, CA, 1995

[13] N. Tredennick: The Case for Reconfigurable Computing; Microprocessor Report, 10,10 (5 Aug 1996),

[14] N. Tredennick: Technology and Business: Forces Driving Microprocessor Evolution; Proc. IEEE 83, 12 (Dec 1995)

[15] A. DeHon: Reconfigurable Architectures for General Purpose Computing; report no. AITR 1586, MIT AI Lab, 1996

[16] R. Kress et al.: A Datapath Synthesis System for the Reconfigurable Datapath Architecture; Asia and South Pacific Design Automation Conference 1995 (ASP-DAC'95), Chiba, Japan, Aug. 29 - Sept. 1, 1995

[17] R. Kress: A fast reconfigurable ALU for Xputers; Ph. D. dissertation, Kaiserslautern University, 1996

[18] D. Auvergne et al.: Power and Timing Modeling and Optimization of Integrated Circuits; ITpress Verlag 1993

[19] Soon Ong Seo: A High Speed Field-Programmable Gate Array Using Programmable Minitiles; Master Thesis, Univ. of Toronto, 1994

[20] R. Hartenstein, J. Becker, M. Herz, U. Nageldinger: A General Approach in System Design Integrating Reconfigurable Accelerators; Proc. IEEE 1996 Int'l Conference on Innovative Systems in Silicon Oct. 9-11,1996 (ISIS'96); Austin, TX, USA

[21] R. Kress: The Software Gap; Proc. 4th Reconfigurable Architectures Workshop (RAW-97; in conjunction with 11th Int'l. Parallel Processing Symposium, IPPS'97), Geneva, Switzerland, April 1-5,1997; in [50]

[22] J. Becker et al.: Custom Computing Machines vs. Hardware/Software Co-Design: From a globalized point of view; Proc. Workshop on Field-programmable Logic and Applications (FPL'96), Darmstadt, 1996

[23] D. Buell, K. Pocek: Proc. IEEE Int'l Workshop on FPGAs for Custom Computing Machines 1993 (FCCM'94), Napa, CA, April 1993

[24] see [23], but also IEEE FCCM-1994, thru -1998

[25] J. Becker: A Partitioning Compiler for Computers with Xputer-based Accelerators; Ph. D. diss., Kaiserslautern 1997.

[26] IEEE annual Conferences on FPGA-based Custom Computing Machines (FCCM); each April, Napa, CA, U.S.A.

[27] K. Buchenrieder: Hardware/Software Co-Design; ITpress Verlag, 1994

[28] R. Hartenstein, A. Hirschbiel, K. Schmidt, M. Weber: A Novel Paradigm of Parallel Computation and its Use to Implement Simple High Performance Hardware; InfoJapan'90- Int'l Conf. memorizing the 30th Anniversary of the Computer Society of Japan, Tokyo, Japan, 1990

[29] R. Hartenstein, A. 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 (1991/92), p. 181-198, (North Holland: invited reprint of [28])

[30] R. Hartenstein, J. Becker, M. Herz, U. Nageldinger: A General Approach in System Design Integrating Reconfigurable Accelerators; Proc. IEEE Int'l. Conf. on Innovative Systems in Silicon, Oct. 9-11, 1996 (ISIS'96); Austin, TX, USA,

[31] R. Hartenstein, J. Becker, M. Herz, R. Kress, U. Nageldinger: A Parallelizing Programming Environment for Embedded Xputer-based Accelerators; High Performance Computing Symposium '96, Ottawa, Canada, June 1996

[32] M. Herz, et al.: A Novel Sequencer Hardware for Application Specific Computing; Proc. 11th Int'l. Conf. on Application-specific Systems, Architectures and Processors, (ASAP'97), Zurich, Switzerland, July 14-16, 1997

[33] R. Hartenstein (invited paper): High-Performance Computing: Über Szenen und Krisen; GI/ITG Workshop on Custom Computing, Schloß Dagstuhl, Germany, June 1996

[34] R. Hartenstein, (invited paper & opening key note): Custom Computing Machines - An Overview; Workshop on Design Methodologies for Microelectronics, Smolenice Castle, Smolenice, Slovakia, Sept. 1995

[35] R. Kress et al.: Customized Computers: a generalized survey; Proc. Workshop on Field-programmable Logic and Applications (FPL'96), Darmstadt, Germany, 1996

[36] R. Hartenstein, A. Hirschbiel, M. Riedmueller, K. Schmidt, M. Weber: A Novel ASIC Design Method based on a Machine Paradigm; IEEE J-SSC 26,7 (July 1991), p. 975-989

[37] R. Kress: Structural Programming with Reconfigurable Data Paths; ITpress Verlag 1998

[38] J. Becker: Reconfigurable Accelerators; Parallelizing Co-Compilation of Structurally Programmable Devices; ITpress Verlag 1998

[39] R. Hartenstein, J. Becker: A Two-level Co-Design Framework for data-driven Xputer-based Accelerators; published in Proc. of 30th Annual Hawaii Int'l Conf. on System Sciences (HICSS-30), Jan. 7-10, Wailea, Maui, Hawaii, 1997

[40] K. Schmidt: A Program Partitioning, Restructuring, and Mapping Method for Xputers; Ph.D. diss., Kaiserslautern 1994

[41] A. Ast, J. Becker, R. Hartenstein, R. Kress, H. Reinig, K. Schmidt: Data-procedural Languages for FPL-based Machines; 4th Int. Workshop on Field Programmable Logic and Applications., FPL'94, Prague, Sept. 7-10, 1994, Springer, 1994

[42] A. Ast et al.:Data-procedural Languages for FPL-based Machines; in: R. Hartenstein, M. Servit (editors): Proc. Int'l Workshop on Field-Programmable Logic and Applications, Prague, Czech Republic, Sept. 1994 (FPL-94), Springer Verlag, LNCS-849

[43] R. Hartenstein, J. Becker, K. Schmidt: Performance Evaluation in Xputer-based Accelerators; Proc. 4th Reconfigurable Architectures Workshop (RAW-97; in conjunction with 11th Int'l. Parallel Processing Symposium, IPPS'97), Geneva, Switzerland, April 1-5,1997; in [50]

[44] L. Lamport: The Parallel Execution of Do-Loops; Communications of the ACM, Vol. 17, No. 2, p. 83-93, Febr. 1974

[45] D. Loveman: Program Improvement by Source-to-Source Transformation; Journal of the Association for Computing Machinery, Vol. 24,No. 1, pp.121-145, January 1977

[46] U. Nageldinger: Design and Implementation of a menu-driven Run Time System for the MoM-3; Master Thesis, Kaiserslautern, 1995

[47] N. Park, A. Parker: Sehwa: A Software Package for Synthesis of Pipelines from Behavioral Specifications; IEEE Transactions on Computer-Aided Design, vol. 7, no. 3, pp. 356-370, March 1988

[48] G. De Micheli: Synthesis and Optimization of Digital Circuits; McGraw-Hill, Inc., New York, 1994

[49] R. Hartenstein, R. Kress: A Scalable, Parallel, and Reconfigurable Datapath Architecture; Sixth International Symposium on IC Technology, Systems & Applications, ISIC'95, Singapore, Sept. 6-8, 1995

[50] R. Hartenstein, K. Prasanna: Reconfigurable Architectures; High Performance by Configware; ITpress Verlag 1997

[51] K. Kennedy: Automatic Translation of Fortran Programs to Vector Form; Techn. Rep. 476-029-4, Rice University, Houston, TX, Oct. 1980