Reiner W. Hartenstein, Karin Schmidt: Combining Structural and Procedural Programming by Parallelizing Compilation; Proceedings of the Symposium on Applied Computing, Nashville, TN, Feb. 1995

COMBINING STRUCTURAL AND PROCEDURAL PROGRAMMING BY PARALLELIZING COMPILATION

Reiner W. Hartenstein, Karin Schmidt
University of Kaiserslautern, Dept. of Computer Science

Keywords: data-parallel machine, field-programmable logic, structural programming, procedural programming

er 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.

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 are not reflecting 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 fulfill 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 (re-) 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 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 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).

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. 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 three major parts: (1) the *data sequencer* with seven *generic address generators* (GAGs), (2) the *reconfigurable ALU* (rALU) with seven *scan windows and several subnets*, and (3) the *data memory* (see figure 1).

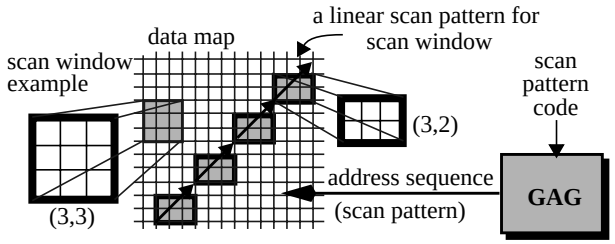

Figure 2 : Different scan window sizes and an address sequence for a linear scan pattern

.In contrast to the one-dimensional von Neumann memory space, the Xputer's *data memory* is primarily organized 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 generic 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 holds data 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. Thus the data sequencer represents the main control part of an Xputer. It consists of seven *generic address generators* (GAGs) operating in parallel and a control logic, 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 (figure 2).

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.

The Compilation Method

A partitioning, restructuring, and mapping method is needed to translate a sequential C program into code which can be executed on an Xputer. This paradigm switch shall be performed without 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.

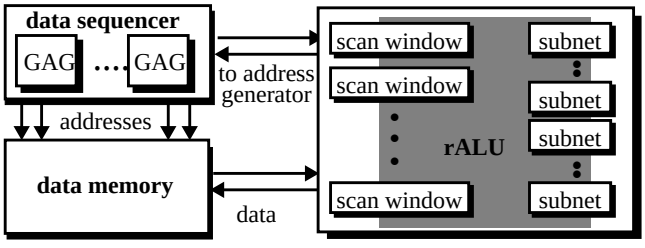

Figure 1 : Xputer block diagram

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). Result of this step is the determination of a partial execution sequence. Secondly the program's data has to be mapped in a regular way onto the 2-dimensionally organized Xputer data map, followed by a computation of the right address accesses (data sequencing) for each variable. 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.

Determination of a Program Execution Sequence

The 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]).

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

$$\pi : N \rightarrow N_i \text{ , } 1 \text{ i n,}$$

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

$$N_k^{\pi} = \{(n \in N) | (\pi(n) \rightarrow N_k)\}$$

and the arc set

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

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

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

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 and building the corresponding equivalence classes which represent convex subgraphs. The definition of a partial order relation for the equivalence classes then gives the execution sequence.

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

$$a \sim_{\text{connect}} b \leftrightarrow (a) = (b) \text{ , } a, b \in N$$

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

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

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

$$N / connect == \{[a]_{connect} | (a \in N)\}$$

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.

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

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

It corresponds to the set of arcs E^*. The presented method for control flow partitioning results in a coarse grained block sequence with a partial execution order. The blocks are still target-hardware independent.

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 has to be 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 described earlier. Each of the subgraphs is then maximally vectorized by using the *Allen-Kennedy Vectorization Algorithm* ([3]). The result is a new sequence of maximally parallelized blocks containing a partial execution order. Vectorization then 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 VF* is introduced. This factor is responsible to subdivide the generated vectors into smaller parts such that an optimal target hardware exploitation can be achieved.

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 optimally exploits the target hardware resources. This is the key for achieving a high performance during Xputer program execution.

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.

Planarization. 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

$$dmap:\ I_i^A \rightarrow \{DM_x, DM_y\}\ , \text{with } 1 \quad i \quad n.$$

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. Xputer dimension mapping is a kind of planarization.

Function dmap. Given is an n-dimensional array A with its according ranges V_j, one for each index I_j, 0 j n-1. The mapping of array A can be performed dimension after dimension by the following recursive function:

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

$$dmap\ (I_0^A) = V_0^A$$

$$dmap\ (I_1^A) = V_1^A$$

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: $j = \begin{cases} n-2, & \text{if n odd} \\ n-1, & \text{if n even} \end{cases}$

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 (seven) 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. 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.

Array Alignment. Let A and $B_1,\ldots, B_n$ denote arrays. *Alignment* of array A with the arrays B_k, 1 k n, means, that the mapped data space of A, called DM_A, is related with all data spaces of B_k, DM_{Bk}. This relation is formulated as:

$$\text{align: } DM_A \times DM_{B_1} \times \ldots \times DM_{B_n} \rightarrow DM_{AB}$$

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*. Both issues are not treated here for lack of space.

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 loader.

Determination of the Data Sequencing

The accessing of the program data variables by their indices is needed for the generation of scan patterns from which the parameter sets for the data sequencer are derived. 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, the alignment, the index expressions, and the according loop limits (upper, lower, step width).

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* can be of an n-dimensional form "for $i_m = l_m$ to u_m", these basis parameter set has to be changed after each completion of a videoscan. The values are given by f_m*_lower*, f_m*_upper*, and *step_*f_m, l k n-1, 2 l 4, and handled by the control part of the data sequencer. In the best case four nested loops can be directly performed without any further control interaction by one scan pattern. All values of an access sequence for a variable *a* providing random index expressions have to be computed according to the following equations.

Access Sequences. 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} the outermost loop and i_0 the innermost loop, and the n-dimensional array variable a $[f_{n-1}(i_{n-1}),\ldots, f_1(i_1), f_0(i_0)]$. The index function f_k is of random or linear form. The ranges of variable "a" are defined by a $[V_{n-1}]\,[V_{n-2}]\,\ldots\,[V_1]\,[V_0]$. The parameters for the concept of an access sequence AS can then be computed by:

upper_x (val = u_0), **lower_x** (val = l_0):

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

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

step_x:

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

step_y:

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

upper_y (val = u_0), **lower_y** (val = l_0):

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

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

f_m_upper (val = u_k), **f_m_lower** (val = l_k):

$$(f_m(i_{n-1},\ldots, \text{val},\ldots, i_0) - c) * \text{dmap}\,(I^a_m) / V_m$$

$$\text{for } i_k \in f_m, \text{ if } m \in \{0,1\} \text{ then c = 0 else 1}$$

step f_m:

$$\frac{\partial}{\partial i_k} f_m(i_{n-1},\ldots, i_k,\ldots, i_0) * \text{dmap}\,(I^a_m) / V_m \quad \text{for } i_k \in f_m$$

These equations are computed for only one special word (handle) in one scan window. All other words have to be addressed by their offsets to the handle, remaining unchanged during a scan pattern. The equations have to be adapted for the case of aligned variables or variables being vectorized.

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 structural and procedural code for the MoM 3 hardware. The *hardware image file* containing the structural information for the configuration of the rDPA ([8]) consists first of a set of scan window definitions together with their types (e.g. integer, float). The second part solely consists of a sequence of assignments, which can be computed in parallel. The *software image file* for the data sequencer hardware comprises an initialization part (e.g. memory size and segments, number and names of the rALU subnets), a load and write strategy for the data of every scan window being used, and the parameter sets for the data sequencing of every GAG.

Conclusions

The paper has sketched a new data-parallel machine, the Xputer, and proposed a parallelizing compilation method for this machine. The method combines structural and procedural programming, according to the Xputer paradigm of a data sequencer hardware and an FPGA-based reconfigurable ALU. Due to the data sequencing, avoiding 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 C as input language) 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 high hardware exploitation of the available resources. This fact guarantees high acceleration gains. The implementation of the proposed compilation method is completed and the performance factors are currently being evaluated.

For further informations, please contact: R.W. Hartenstein, University of Kaiserslautern, Department of Computer Science, P.O Box 3049, 67653 Kaiserslautern, Germany, email: abakus@informatik.uni-kl.de.

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.Sehti, 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.