Memory Access Optimization and RAM Inference for Pipeline Vectorization

M. Weinhardt and W. Luk

© Springer-Verlag Berlin Heildelberg 1999. This paper was first published in Field-Programmable Logic and Applications, Proceedings of the 9th International Workshop, FPL '99, Lecture Notes in Computer Science 1673, Springer-Verlag 1999, ISBN 3-540-66457-2, pp. 61–70. Reproduced with the permission of Springer-Verlag.

http://www.springer.de

Memory Access Optimization and RAM Inference for Pipeline Vectorization*

Markus Weinhardt and Wayne Luk

Department of Computing, Imperial College, London, UK
{mw8, wl}@doc.ic.ac.uk

Abstract. This paper describes memory access optimization in the context of pipeline vectorization, a method for synthesizing hardware pipelines in reconfigurable systems from software program loops. Since many algorithms for reconfigurable coprocessors are I/O bound, the throughput of the coprocessor is determined by the external memory accesses. Thus access optimizations directly improve the system's performance. Two kinds of optimizations have been studied. First, we consider methods for reducing the *number* of accesses based on saving frequently-used data in on-chip storage. In particular, recent FPGAs provide on-chip RAM which can be used for this purpose. We present *RAM inference*, a technique which automatically extracts small on-chip RAMs to reduce external memory accesses. Second, we aim to minimize the *time* spent on external accesses by scheduling as many accesses in parallel as possible. This optimization only applies to architectures with multiple memory banks. We present a technique which allocates program arrays to memory banks, thereby minimizing the overall access time.

1 Introduction

The reconfigurable computing community is increasingly interested in high-level compilers which translate a software program into both machine code for a host processor and configuration bitstreams for a field-programmable accelerator [1–3]. This work has shown that loops are the most promising candidates for acceleration. The coprocessors must also exploit the hardware's parallelism to achieve a sufficient speedup. Our approach, called *pipeline vectorization* [1, 4, 5], achieves this by vectorizing loops and synthesizing pipeline circuits. Many image processing and DSP applications exhibit no or little dependences which limit pipelining. Consequently, provided that enough FPGA resources are given, only memory accesses limit the throughput of the pipelines: the coprocessors are I/O bound. We present techniques that reduce these access times, thus directly improving system performance. One of the techniques, the use of shift registers, was first suggested in [6] for pipelines in reconfigurable systems. However, the

* This work is supported by a European Union training project financed by the Commission in the TMR programme, the UK Engineering and Physical Sciences Research Council, Embedded Solutions Ltd., and Xilinx Inc.

registers were declared explicitly and not synthesized automatically as in our approach.

Since FPGA-based accelerators typically organize external memory as one or more fast local SRAM banks accessible from the host and the FPGAs [7–9], we also address the allocation of data to memory banks. Gokhale and Stone [10] independently developed an allocation algorithm based on a technique called implicit enumeration. This algorithm is slightly faster, but less general than our approach.

The remainder of this paper first reviews pipeline vectorization. Then, Section 3 defines access equivalence and explains how shift registers synthesized for equivalence classes reduce the number of memory accesses. Next, Section 4 introduces RAM inference, which automatically extracts small on-chip RAMs by analyzing access equivalence in nested loops. Section 5 presents a technique to allocate program arrays on multiple memory banks. Finally, performance results are given in Section 6 before we conclude the paper in Section 7.

2 Pipeline Vectorization

We outline pipeline vectorization using a computationally expensive running example, image skeletonization [11]. It iteratively performs erosion, dilation, difference and OR operators on an input image, thereby producing its "skeleton". Figure 1 outlines the program. Note that successive versions of the input image and of the resulting skeleton are copied back and forth between `image1/skeleton1` and `image2/skeleton2` during the iterations of the outer `while` loop.

Pipeline vectorization consists of three main steps: candidate loop selection, dataflow graph generation, and pipelining. In the first step, candidates for hardware acceleration are selected. These are inner loops which do not contain non-synthesizable functions and are normalized so that their index variables are initialized to zero and incremented by one. In the example in Figure 1, the innermost `for` loop (line 12 to line 23) is a candidate. Note that the given form of the loop was generated by a transformation [1] which merged the originally independent operators so that they are combined in one loop and can be synthesized into a single coprocessor. The loop contains the functions `Fmin` and `Fmax` (not elaborated in Figure 1) which perform the bulk of the program's computation: they respectively compute the minimum or maximum of 21 pixels of an image (a 5×5 window without the corners). Loops spanning these 5×5 windows are unrolled by another transformation [1] to remove the internal loops in the functions.

Next, a dataflow graph is generated for the loop by analyzing the data dependences within the loop body. For conditionally assigned variables, multiplexers are inserted to select the correct values. We allocate scalar input and output variables to FPGA registers. Array variables are stored in local (off-chip) memory, since it is expensive to store large data arrays, such as image data, on an FPGA. Figure 2 shows the result for the example. The internal structure of the blocks `Fmin` and `Fmax`, the conditions guarding some operations, and the logic

```
 1  char image1[N][N], image2[N][N], skeleton1[N][N], skeleton2[N][N];
 2  char *im_in, *im_out, *skel_in, *skel_out; ...
 3  ... /* read input image to image1 and reset skeleton1 */
 4  pixel_on = true; count = 0;
 5  while (pixel_on) {
 6    pixel_on = false; count++;
 7    im_in    = (count % 2 == 1) ? image1    : image2;
 8    im_out   = (count % 2 == 1) ? image2    : image1;
 9    skel_in  = (count % 2 == 1) ? skeleton1 : skeleton2;
10    skel_out = (count % 2 == 1) ? skeleton2 : skeleton1;
11    for (x=0; x<N+2; x++)
12      for (y=0; y<N+1; y++) { /*** candidate loop ***/
13        if (x<N && y<N)
14          im_out[x,y] = (y<2 || y>=N-2 || x<2 || x>=N-2) /* A */
15                        ? 0 : Fmin(im_in,x,y);
16        if (x>=2 && y>=1) {
17          filter      = (y<3 || y>=N-1 || x<4 || x>=N)   /* B */
18                        ? 0 : Fmax(im_out,x-2,y-1);
19          pixel = im_in[x-2,y-1] - filter;
20          skel_out[x-2,y-1] = skel_in[x-2,y-1] | pixel;
21          if (pixel==255) pixel_on = true;
22        }
23      }
24  }
```

Fig. 1. Skeletonization program.

for updating `pixel_on` are omitted for clarity. `Fmin` and `Fmax` each contain 20 comparators and multiplexers connecting the inputs.

Before coprocessor operation, scalar input variables must be loaded into the FPGA registers, and input arrays copied to the local memory. During operation, values from the arrays whose index depends on the loop index (which we call vector inputs) must be read, and vector outputs written in every iteration. In our example, 42 values have to be read and two written for each iteration computing a pixel. We refer to these inputs and outputs collectively as *vector ports* of the circuit. Though the dataflow graph is acyclic and thus could be pipelined and produce an output in every clock cycle, the performance of the circuit is very low because of the memory accesses.

Additionally, dependences carried by the candidate loop can prevent legally pipelining the dataflow graph and must therefore be analyzed. Our example loop contains such dependences from statement A which writes `im_out[x,y]` (line 14 in Figure 1) to statement B (line 18) which reads this value in subsequent iterations (as `im_out[x,y-1]` and `im_out[x,y-2]`). However, these are regular dependences (with constant dependence distances) which do not prevent pipelining but require some changes to the dataflow graph. They will be explained for our example in the next section.

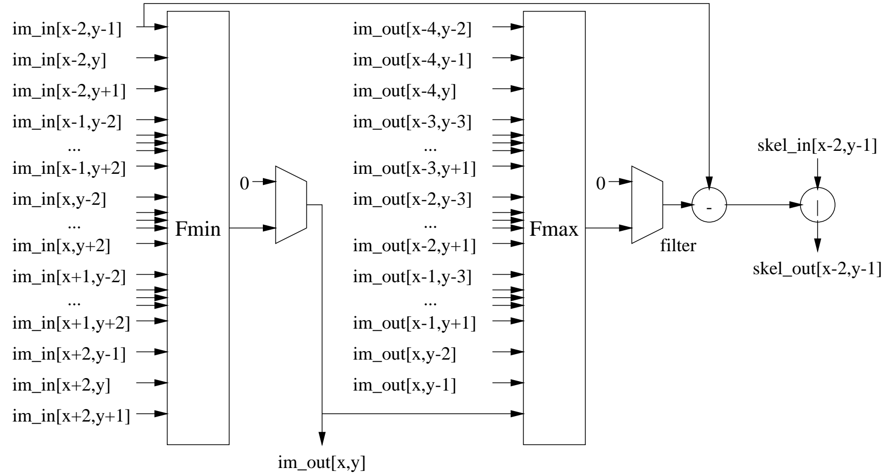

Fig. 2. Dataflow graph for Figure 1.

In contrast to classical vectorization, we do not explicitly generate vector instructions. Rather all instructions of the loop body are vectorized implicitly by pipelining the dataflow graph.

3 Access Equivalence and Shift Registers

It is obvious that the vector inputs in Figure 2 access many pixels of `im_in` and `im_out` repeatedly in successive loop iterations. These redundant reads can be avoided by combining vector ports and storing reused pixels in FPGA registers. To do this automatically, we define classes of *equivalent vector accesses* which access the same array element in different loop iterations.

Definition 1 (Equivalence of Vector Accesses) *Two vector accesses differing only in one index are* equivalent with respect to index variable I *iff the indices have the form* $C_1 + S \times I$ *and* $C_2 + S \times I$*, where* S*,* C_1 *and* C_2 *are constants,* and C_1 *mod* $S = C_2$ *mod* S*. In other words, the indices are linear functions of* I *with the same "stride"* S*, differ only in the constant term, and the constants are in the same residue class modulo* S*. The accesses'* distance *is* $D = (C_1 - C_2)/S$*. All equivalent vector accesses form an* equivalence class*.*

For instance, input ports `im_in[x-2,y-1]`, `im_in[x-2,y]` and `im_in[x-2, y+1]` in Figure 2 differ only in the second index and have the form $C + S \times I$ (with $S = 1$, $I =$ `y` and $C = -1$, 0 and 1, respectively). Since they are all equal *mod* 1, they form an equivalence class with respect to the candidate loop's index variable `y`. In contrast, accesses `A[2*I]` and `A[2*I-1]` are not equivalent with respect to `I`. The index has the form $C + S \times I$ for both accesses (with $S = 2$), but the modulo condition is not met. This shows that the expressions can never access the same array element.

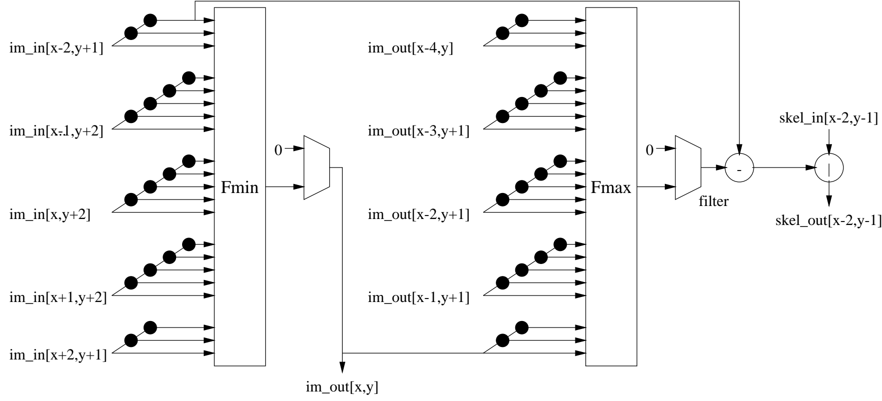

Fig. 3. Circuit with shift registers.

Ports for equivalent vector accesses with respect to the candidate loop's index variable can be combined. Thus only one vector port is necesssary for an entire access equivalence class. The vector input with the largest C (for $S > 0$) or the smallest C (for $S < 0$) for each class is retained and connected to the input of a shift register. Its length is chosen as the largest distance D to any other input in its class. Then every other input in the class is substituted by an access to the value delayed by the distance D for that input in the shift register.

In Figure 2, input port `im_in[x-2,y+1]` is retained and connected to a shift register of length two, since the largest distance D is two (between `im_in[x-2,y-1]` and `im_in[x-2,y+1]`). Input ports `im_in[x-2,y]` and `im_in[x-2,y-1]` are substituted by connections to the shift register. Figure 3 shows the resulting circuit. Registers are represented as black dots. All inputs to the other image lines are combined similarly; and the inputs for `im_out[x]` even use the values produced by `Fmin`. So no vector input for `im_out[x]` is necessary at all, and the dependences mentioned in Section 2 which prohibited pipelining the dataflow graph are removed. Thus memory reads, the actual computation, and memory writes can be pipelined. Since the circuit is acyclic, the combinational delay could be reduced to the delay of a single logic cell or FPGA look-up table by pipelining. However, the throughput is still limited by the time required for the remaining 10 vector read and two vector write accesses per loop iteration. Obviously, additional time is needed for filling the shift registers and the pipeline stages. As long as we have moderately long loops, the overhead is outweighed by the reduced time for each loop iteration. We have implemented the combination of input registers in an earlier prototype compiler [4, 5].

4 RAM Inference

The use of shift registers presented in the previous section greatly reduces the number of memory accesses. However, the method is restricted to data reuse

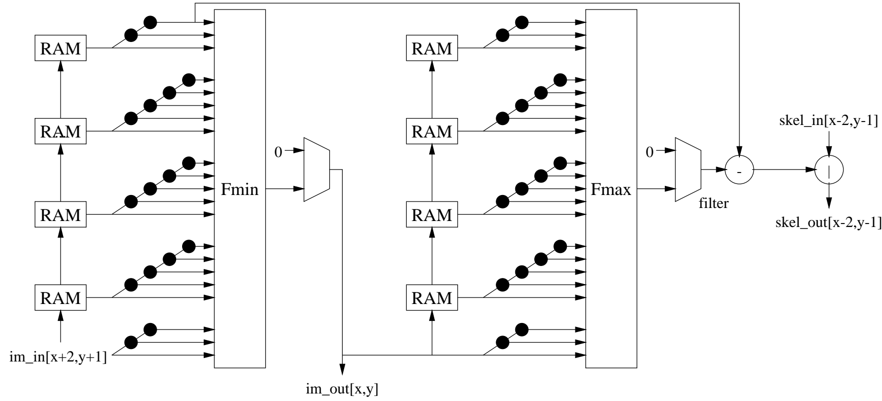

Fig. 4. Circuit with shift registers and on-chip RAMs.

within one pipeline execution, namely within the candidate loop. Further reductions are possible if data accessed in a previous pipeline execution can be reused. We have to store entire slices of an array on-chip. Since this is infeasible with shift registers built from ordinary FPGA registers, we utilize on-chip RAM available on newer FPGA architectures like Xilinx XC4000, Virtex or Altera FLEX10K instead.

Analysis of the next outer loop of a candidate determines which redundant accesses can be removed. We call this process *RAM inference* since it synthesizes small RAMs from a program like HDL synthesis systems use register inference to determine which variables need to be stored in registers. RAM inference is only legal if non-candidate code in the next-outer loop does not alter the arrays in question. If this condition is met, we determine equivalence classes according to Definition 1, but this time with respect to the next-outer loop's index variable. In every equivalence class only one access is necessary. The others are substituted by accesses to new on-chip RAMs which store all values accessed during an entire pipeline execution. The transformation is analoguous to the synthesis of shift registers, but instead of a register an entire small on-chip RAM bank is synthesized. In our example program, the outer `for` loop (index `x`) does not alter any arrays outside the candidate loop and carries no dependences, and all accesses to `im_in` and `im_out` are in respective equivalence classes. Thus only `im_in[x+2,y+1]` needs to be retained. All older image lines are stored in separate on-chip RAM banks which can be accessed concurrently; the RAM banks serve as delay lines. All accesses to `im_out` can directly use stored output values from `Fmin` as well. Thus the number of vector inputs is further reduced to two. Figure 4 shows the resulting circuit. The RAM control logic is omitted for clarity.

Though RAM inference can provide big speedups, it also uses many FPGA resources, especially if the candidate loops are long and consequently large RAMs are needed. Thus a time/area trade-off arises. A hardware estimater must deter-

mine the number of RAMs of the required size which fit on the given resources, and select an appropriate implementation.

5 Array Allocation

The previous sections introduced techniques to reduce the required number of external memory accesses. However, the pipeline's throughput is actually determined by the *time* required for these accesses. On architectures with a single memory bank, this time is equivalent to the number of vector accesses or ports. For architectures with multiple memory banks, the throughput can be increased by scheduling as many accesses as possible in parallel. By distributing the ports over the memory banks, the maximal number of accesses required per iteration in one single bank can be minimized.

Since we do not attempt to distribute arrays over several memory banks, all accesses to the same array necessarily have to be sequentialized. We therefore aim to allocate a program's arrays to memory banks such that the resulting accesses are as evenly distributed as possible. The banks must also be long enough to hold the allocated arrays.[1] This is represented by the following problem specification. It applies to a single candidate loop.

Problem 1 (Simple Array Allocation) *Allocate m arrays $A_1, ..., A_m$ of lengths $l_1, ..., l_m$ and with $p_1, ..., p_m$ ports (accesses in one candidate loop iteration) to n memory banks $B_1, ..., B_n$ of size $s_1, ..., s_n$ such that the maximal number of ports allocated to a bank (via the arrays) is minimal and the arrays fit on the banks, that is find a mapping $M = [1:m] \rightarrow [1:n]$ which mimimizes*

$$maxp = \max_{i \in [1:n]} \left(\sum_{j \in [1:m], M(j)=i} p_j \right)$$

and fulfills the following constraints:

$$\forall i \in [1:n] : \sum_{j \in [1:m], M(j)=i} l_j \leq s_i$$

Unfortunately this simple problem specification is inaccurate for many programs since accesses can refer to different arrays in different instances of a candidate loop, which we call *situations*. In the program in Figure 1, for instance, accesses to `im_in` can refer to either `image1` or `image2`. Counting both references would greatly overestimate the access numbers and would miss the information that `image1` can either serve as `im_in` or `im_out`, but never as both at the same time. So it is useful to distinguish situations whenever possible, as those for odd and even values of `count` in the example program. We extend problem 1 accordingly.

[1] We do not consider differing memory widths. They could easily be modelled as additional constraints for the allocation.

Problem 2 (Simultaneous Array Allocation for Situations) *Allocate m arrays to n memory banks as in Problem 1. Let p_j^k be the number of ports for array A_j in* situation S_k, *for situations $S_1, ..., S_o$. A simultaneous allocation for all situations must minimize $maxp$ defined as follows:*

$$maxp = \max_{k\in[1:o]} \left(\max_{i\in[1:n]} \left(\sum_{j\in[1:m], M(j)=i} p_j^k \right) \right)$$

We formulate the more general Problem 2 as an *integer linear program (ILP)* and use a standard ILP solving algorithm to find an optimal solution. Although this algorithm is potentially exponential, it works efficiently for the problem at hand since the number of memory banks, arrays and situations found in practical applications is typically less than five. The ILP determines the maximum access number $maxp$ and the 0-1 variables $x_{j,i}$ for $j \in [1:m]$ and $i \in [1:n]$ which determine the mapping M: $x_{j,i} = 1 \Leftrightarrow M(j) = i$. The cost function is simply $maxp$ which must be minimized. The solution is subject to the following constraints:

$$\forall k \in [1:o], \forall i \in [1:n] : \sum_{j\in[1:m]} p_j^k \times x_{j,i} \leq maxp \quad (1)$$

$$\forall j \in [1:m] : \quad \sum_{i\in[1:n]} x_{j,i} = 1 \quad (2)$$

$$\forall i \in [1:n] : \quad \sum_{j\in[1:m]} l_j \times x_{j,i} \leq s_i \quad (3)$$

Constraints (1) ensure that $maxp$ is met for all banks and situations. Constraints (2) guarantee that M is a function, while (3) represent the memory length constraints.[2]

The example circuit with shift registers (Figure 3) provides the following input values: $p_{image1} = 5$, $p_{image2} = 5$, $p_{skeleton1} = 1$, $p_{skeleton2} = 1$ for both situations (`count % 2 = 1` and `count % 2 = 0`). Given that two memory banks of sufficient size ($s \geq 2 \times N \times N$) are available, `lp_solve` [12] allocates `image1` and `skeleton1` to B_1 and `image2` and `skeleton2` to B_2. $maxp = 6$, thus the 12 accesses are evenly distributed in both situations.

This method can be extended to find a single array allocation for an entire program by combining constraints for the set L of all loops moved to coprocessors in a single ILP. The cost function must be extended to $\sum_{l\in L} w_l \times maxp_l$ where w_l is the estimated iteration count for loop l during the entire program execution. This approach is taken in [10]. A combined allocation has the advantage that data need not be rearranged between coprocessor executions, but it may not be optimal for every single coprocessor. However, if host access to the data is required between the executions, the data have to be copied back and forth anyway. Hence optimal independent allocations should be used in some cases, depending on a dataflow analysis. To make the situation worse, the set L cannot

[2] Note that dual-ported memory can be easily modelled. Constraints (1) must be replaced by $\frac{1}{2}\sum_{j\in[1:m]} p_j^k \times x_{j,i} \leq maxp$ if B_i is dual-ported, that is accommodates twice as many accesses as single-ported memory.

Implementation	A - no optim. (Figure 2)	B - shift reg.s (Figure 3)	C - shift reg.s + RAMs (Figure 4)
Memory accesses	44	12	4
Access cycles with 1 bank	44	12	4
Access cycles with 2 banks	22	6	2
Access cycles with 4 banks	21	5	1

Table 1. Performance for skeletonization program.

be determined before array allocation since it depends on the achieved performance for a loop. Hence the dataflow analysis and the estimates for w_l required for optimal allocation have to be combined with hardware/software partitioning [13]. This is beyond the scope of this paper.

6 Results

Table 1 summarizes the performance characteristics of the skeletonization program in Figure 1. There are three implementations: without optimizations (A), with shift registers (B), and with shift-registers and small RAMs (C) on systems with one, two and four memory banks. Obviously implementations without any optimizations (A) are infeasible because they require over 20 access cycles in any case. Shift registers reduce the number to between 5 and 12 cycles, depending on the number of memory banks. RAM inference yields another cycle reduction — and hence execution speedup — by a factor three (for one or two banks) or five (for four banks). Even with optimal allocation of the arrays, the number of access cycles does not linearly fall with the number of memory banks, because accesses with the same arrays are bound to one bank. This is why moving from two to four banks for implementation B only speeds up the coprocessor by 17 % rather than 50 % as for implementation C.

We implemented versions B and C on an RC1000-PP [9] board with two memory banks in Handel-C [14]. For a circuit running at 20 MHz, we measured 6 ms for one iteration of the outer `while` loop on a 128×128 image for implementation B, and 2 ms for implementation C — a speedup factor of three as expected. We measured 65 ms per iteration for software execution on a 300 MHz Pentium PC. Thus implementation C with two memory banks is about 32 times faster than software, and the anticipated speed for four banks (1 ms) is 65 times faster. However, there is a small overhead per image for transferring the data (2 ms). Since a typical image needs around 15 to 30 skeletonization iterations, the impact of this overhead is relatively small. Configuring the Xilinx XC4085 FPGA takes 780 ms on the RC1000-PP; only one configuration is required during the entire program execution.

7 Conclusion and Future Work

We have discussed the crucial impact of external memory accesses on the performance of I/O bound reconfigurable coprocessors. We presented methods to synthesize shift registers and small on-chip RAMs to reduce significantly the number of accesses. A technique to allocate arrays to memory banks has also been developed, which minimizes the overall memory access time. We used an extended example to demonstrate the effects of our optimizations. Our approach has been validated by a prototype implementation on the RC1000-PP board.

We are currently including the methods described here in our pipeline compiler based on the SUIF compiler framework [15]. Our methods can be used to extend hardware compilers such as Handel-C as well. Future work includes evaluating different types of on-chip memory in the latest FPGA families. Further investigations on the area/time trade-offs of RAM inference is also required to automatically select an optimal implementation.

References

1. M. Weinhardt and W. Luk. Pipeline vectorization for reconfigurable systems. In *Proc. FCCM'99*. IEEE Computer Society Press, 1999.
2. G. Haug and W. Rosenstiel. Reconfigurable hardware as shared resource in multipurpose computers. In *Proc. FPL'98*. Springer, 1998.
3. M. B. Gokhale and J. M. Stone. NAPA C: compiling for a hybrid RISC/FPGA architecture. In *Proc. FCCM'98*. IEEE Computer Society Press, 1998.
4. M. Weinhardt. Compilation and pipeline synthesis for reconfigurable architectures. In *Reconfigurable Architectures Workshop RAW'97*, 1997.
5. M. Weinhardt. *Übersetzungsmethoden für strukturprogrammierbare Rechner (Compilation techniques for structurally programmable computers, in German).* PhD thesis, Universität Karlsruhe, July 1997.
6. S. A. Guccione and M. J. Gonzalez. A data-parallel programming model for reconfigurable architectures. In *Proc. FCCM'93*. IEEE Computer Society Press, 1993.
7. S. Nisbet and S. A. Guccione. The XC6200DS development system. In *Field Programmable Logic and Applications*, LNCS 1304, 1997.
8. D. A. Buell, J. M. Arnold, and W. J. Kleinfelder. *Splash 2 - FPGAs in a Custom Computing Machine*. IEEE Computer Society Press, 1996.
9. Embedded Solutions Limited. *RC1000-PP Product Information Sheet*. `http://www.embedded-solutions.ltd.uk/ProdApp/RC1000PP.htm`.
10. M. B. Gokhale and J. M. Stone. Automatic allocation of arrays to memories in FPGA processors with multiple memory banks. In *Proc. FCCM'99*. IEEE Computer Society Press, 1999.
11. H. R. Myler and A. R. Weeks. *Computer Imaging Recipes in C*. Prentice Hall, 1993.
12. M. Berkelaar. Unix manual page of `lp_solve`. Eindhoven University of Technology, Design Automation Section, 1992.
13. M. Weinhardt. Integer programming for partitioning in software oriented codesign. In *Proc. FPL'95*. Springer, 1995.
14. Embedded Solutions Limited. *Handel-C Reference Manual*, 1998.
15. The Stanford SUIF Compiler Group. Homepage `http://suif.stanford.edu`.