Correctness Preserving Transformations for the Design of Parallelized Low-Power Systems

Marc Theisen and Felix C. Gärtner

DFG-Graduiertenkolleg ISIA
Darmstadt University of Technology
Karlstraße 15, D-64283 Darmstadt
theisen@mes.tu-darmstadt.de
felix@informatik.tu-darmstadt.de
http://www.microelectronic.e-technik.tu-darmstadt.de/research/isia/

Abstract. With growing shares of the market of mobile microelectronic systems the reduction of energy consumption is becoming a prominent design goal. We present a method to reduce the energy consumed in processor and memory elements. The basic idea of the approach is to apply parallelizing transformations, a method known from compiler construction. In order to be sure that the transformed system still behaves according to the original specification a formal proof of correctness preservation is given.

1 Introduction

During the design process of microelectronic systems, optimizing the area and throughput are not the only goals. A target of increasing importance is to lower the *energy* consumption of the system. With higher integration densities thermal constraints become very important. Thus, it is necessary to reduce the *power* consumed by the circuit. If there are no timing constraints the power can easily be reduced by extending the execution time. Since power is energy divided by time, the dissipated energy is not minimized and this is a central concern for battery driven mobile systems. The goal of this paper is to propose techniques to reduce the energy consumption while at the same time still considering timing constraints. Note that if the execution time is kept unchanged power and energy reduction is equivalent.

Many modern circuits for telecommunication applications are simulated at the system level with the COSSAP tool from SYNOPSYS. Such a system can either be synthesized by a core-based design or with high-level design methods. In this paper optimization methods which can be applied during the high-level synthesis are discussed. An important issue is the optimized implementation of loops. Especially, in algorithms in the area of signal processing and communication technologies loops are the central parts. The problem is that they often are computationally intensive, like for example adaptation algorithms in filters, as well as data intensive, like for example image processing algorithms for mobile communication systems.

In order to reduce the energy consumed in computationally intensive applications has been proposed to reduce the supply voltage [11, 6, 12]. Since the timing constraints

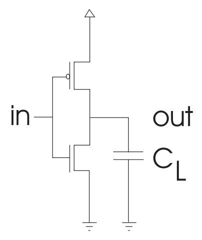


Fig. 1. Inverter

have to be fulfilled the operations have to be executed in parallel. The problem is that data dependences may prevent a parallelization. For removing the dependences it is possible to apply loop and index transformations which are used in parallelizing compilers for parallel computing machines [8, 1–3]. In the case of data intensive algorithms, energy can be gained in the memory elements too. For example, accessing a single 16 bit memory word requires less energy than two separate accesses to two separate 8 bit memory words [10, 4]. Again, this is only possible if there are no limiting data dependences. Therefore, in both cases parallelizing transformations can be used to minimize the dissipated energy. Thereby, it is possible to reduce the power dissipated in the processing and the memory elements simultaneously.

In order to make sure that the transformations used during the high-level synthesis process do not cause any faults in the circuit, simulations are run before and after the synthesis step. These simulations cover only the tested cases and are very time consuming. Therefore, for the central parallelizing transformation discussed in the following, a mechanical correctness proof was performed using the verification tool PVS [9]. By this proof it is assured that after the application of the transformation the circuit works correctly. To the best of our knowledge, we have not seen any other work which combines the above parallelizing transformations for high-level synthesis with a mechanical correctness proof.

In Section 2 and 3 the sources of energy dissipation are analyzed, different computation techniques are discussed and consequences for the system design are drawn. Then in Section 4 these results will be applied and the parallelizing transformations will be introduced. After that in Section 5 a proof for the formal correctness of the transformation is given.

2 Reduction of Energy Consumption in Processing Elements

The total power P_{tot} consumed in a CMOS circuit consists of three components [12] (see Figure 1): the capacitive switching power P_{switch} which is consumed during the charging and discharging of the load capacitance C_L, the power P_{short} consumed during the switching when both NMOS- and PMOS-blocks conduct and the power P_{leak} caused by subthreshold currents. Thus:

$$P_{tot} = P_{switch} + P_{short} + P_{leak}. \tag{1}$$

In comparison to P_{switch} the other two components P_{short} and P_{leak} are so small for current technologies that they can be neglected [11]. The capacitive switching power which is caused by charging and discharging the capacitance C_L can be computed by

$$P_{switch} = \frac{1}{2} C_L V_{dd}^2 \alpha f \tag{2}$$

where V_{dd} is the supply voltage, f the clock frequency and α the average number of transitions per clock cycle. Because of the quadratic influence the most important parameter to reduce the power is the supply voltage V_{dd}. But changing this parameter might influence other characteristics of the system in a negative way. Therefore, a good trade-off has to be found.

As a first order approximation the delay t_d of the circuit elements is given by [5]:

$$\begin{aligned} t_d \sim RC_L &= \frac{C_L V_{dd}}{I} \\ &= \frac{C_L V_{dd}}{\frac{\mu C_{ox}}{2} \frac{W}{L} (V_{dd} - V_T)^2} \end{aligned} \tag{3}$$

where C_L is the load capacitance, R the on-resistance of the transistors and V_T the threshold voltage. In this approximation it is assumed that the transistor is in saturation and that the on-resistance is constant. Equation 3 shows that the delay is proportional to $\frac{V_{dd}}{(V_{dd}-V_T)^2}$ and that it increases drastically with V_{dd} approaching V_T. Another negative consequence is that a reduction of V_{dd} leads to an increased signal-to-noise ratio.

The idea for saving power in the processing elements is to reduce V_{dd} to V_{dd}^{dou} such that t_d doubles. In order not to increase the computing time two processing elements are used instead of only one. Thereby, C_L doubles in Equation (2), but in total P_{switch} will decrease because of the quadratic influence of V_{dd}. If the ALCATEL MIETEC 0.35μ CMOS technology which has been made available to us by EUROPRACTICE is used then the delay doubles if V_{dd} is reduced from originally $V_{dd}^{ori} = 3.3V$ to $V_{dd}^{dou} = 2.1V$ and it triples if V_{dd} is decreased from $3.3V$ to $V_{dd}^{tri} = 1.7V$. The consumed energy can be computed by

$$E_{switch} = \frac{1}{2} C_L V_{dd}^2. \tag{4}$$

Therefore, the relative saving S_{dou} by the usage of two processing elements is given by:

$$S_{dou} = 1 - \frac{2(V_{dd}^{dou})^2}{(V_{dd}^{ori})^2} = 0.19. \tag{5}$$

In the case of three processing elements the saving S_{tri} is:

$$S_{tri} = 1 - \frac{3(V_{dd}^{tri})^2}{(V_{dd}^{ori})^2} = 0.20. \tag{6}$$

This shows that a high improvement is gained by reducing the supply voltage from $3.3V$ to $2.1V$. This can be only slightly improved by a further reduction to $1.7V$.

In order to compare these results, SPICE simulations were run for the above mentioned CMOS technology. The data is given in Table 1 and it can be seen that the voltage can even be reduced to $1.7V$ until the delay doubles and to $1.4V$ until the delay triples. At a voltage of $1.7V$ the gain of energy is 49.48% and at $1.4V$ it is 49.91%. This proves that the results based on the approximation in (2) and (3) are too conservative. This is due to the assumption that the transistor is in saturation and that the on-resistance is constant.

voltage $[V]$	delay $[10^{-10}s]$	energy $[10^{-14}J]$	processing elements	gain [%]
3.3	1.763	10.01	1	0
1.7	3.365	2.5285	2	49.48
1.4	4.784	1.6715	3	49.91

Table 1. Energy reduction

3 Reduction of Energy Consumption in Memory Elements

In the following only on-chip memories and SRAM will be taken into consideration because it needs less power than DRAM. The power consumed in total for modern on-chip SRAM is dominated by the capacitive switching power which can be computed by

$$P = \frac{1}{2} V_{dd}^2 C_{eff} f_{access} \tag{7}$$

where C_{eff} is the effective capacitance and f_{access} the real access rate to the memory [4]. Since the capacitances for read and write operations are different the model based on Equation 7 can be improved by

$$P = \frac{1}{2} V_{dd}^2 (C_{read} f_{read} + C_{write} f_{write}). \tag{8}$$

Following the macromodel of Landman [4] the read and write capacitances can be approximated to the first order by the following equation:

$$C_{mode} = C_{mode,0} + C_{mode,1} W + C_{mode,2} N + C_{mode,3} NW \tag{9}$$

where *mode* is either *read* or *write*, N the bitwidth and W the number of words of the memory. In order to get precise data for the ALCATEL MIETEC 0.35μ CMOS technology we used the *uni2.1* RAM/ROM generator files and the *adsGenerator* of the ALCATEL Microelectronics *ADS* front-end design system for digital circuit design which has been made available to us by EUROPRACTICE within the designkit version 10.00. The data is generated for the low power memory type SPS3. In Figure 2 the dependence of the power/MHz on the word depth is shown for several bit words. Equations 8 and 9 show a linear dependence of the energy consumption on the word depth using constant bitwords. The experimental data shown in Figure 2 support this model for small word depths. However, for large word depths the energy consumption increases due to a changed memory topology. Figure 2 shows that, e.g., for a memory of the word depth

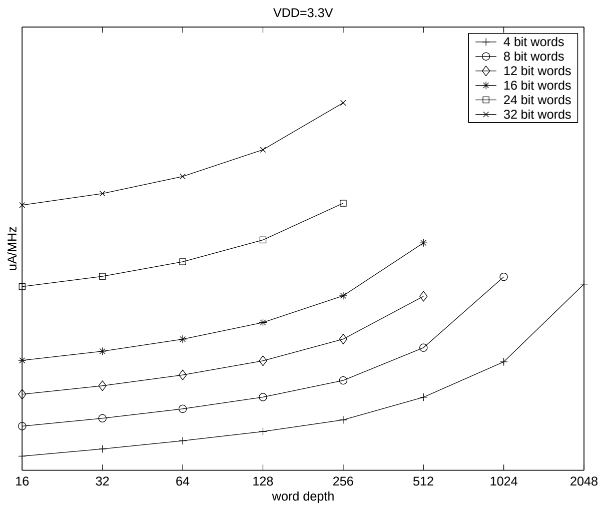


Fig. 2. Memory energy consumption

1024 with 8 bit words an energy of $a\frac{W}{MHz}$ is consumed. Whereas if we implement a memory with half of that word depth, in this case 512, but with 16 bit words then a higher energy of $b\frac{W}{MHz}$ is necessary, but the access frequency can be reduced by half. Since $\frac{1}{2}b\frac{W}{MHz} < a\frac{W}{MHz}$ energy can be saved by doubling the bit words. In Figure 3 the relative improvement of the energy consumption is shown in dependence of the word depth. In this graph the improvement is assigned to the original word depth. It must be noted that the improvement gets smaller for smaller word depth and the doubling of larger bit words. This is due to the relative overhead of the decoder in comparison to the reduced number of memory cells. It can be seen that the relative gain by tripling the bit words is even higher compared to the case in which the bit words are doubled and that the energy can be reduced by this technique up to 50%. It has to be pointed out that this technique is only applicable if both data items which will be mapped to the new common word are read and written at the same time. For many algorithms this method cannot be applied directly, but often the algorithms can be transformed such that this approach can be used to reduce energy consumption. In the next section, high-level transformation methods will be introduced that reduce the supply voltage as described previously and that multiply the bitwidth.

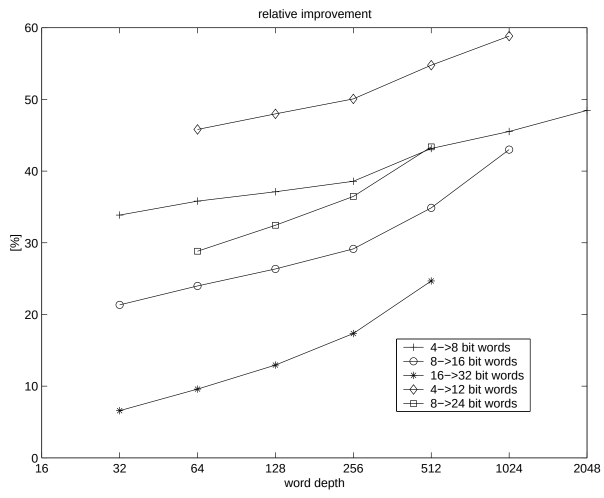


Fig. 3. Reduction of energy by doubling and tripling the bit words

4 Transformations Enabling the Minimization of Energy Consumption

This section describes an automizable method for transformation of a high-level system specification such that both previously introduced techniques can be applied for saving energy. For the application of our method it is required that a perfectly nested loop is given and that the loop limits are known at compile time [13]. Further, the index functions of the loop limits and of the array references are assumed to be affine-linear functions of the indexes of the surrounding loops ($f(x) = ax + b$ is called a linear function if b is equal to 0 otherwise affine-linear). As shown in [1, 2] the data dependences can be extracted from the array references and it is assumed that the dependence distance vectors are uniform over the index space. These conditions are fulfilled for many algorithms in signal processing and filtering. An example fulfilling these conditions is given in Figure 4.

At the beginning of the transformation process the dependences are computed, analyzed and the data dependence matrix is set up. This dependence matrix has to be transformed in such a way that the dependences are resolved in order to be able to parallelize the code. In order to describe the transformation method the example given in Figure 4 will be taken into consideration. As it can be seen in Figure 5 all those index points can be computed in parallel which are on one of the lines called wavefronts. There are several wavefronts possible that differ in the number of time steps and processing elements necessary for the computation. With the help of the equation

$$n_{it} = \lceil \frac{I_{1,upper} - I_{1,lower}}{w_1} + \ldots + \frac{I_{m,upper} - I_{m,lower}}{w_m} \rceil + 1 \qquad (10)$$

```
$L_{1,1}$: **for** $I_1$ in 1 to 4 **loop**
    $L_{1,2}$: **for** $I_2$ in 1 to 4 **loop**
        $H_1$: A($I_1$,$I_2$):=f(A($I_1$-1,$I_2$),A($I_1$,$I_2$-1));
    **end loop**;
**end loop**;
```

Fig. 4. Perfectly nested loop.

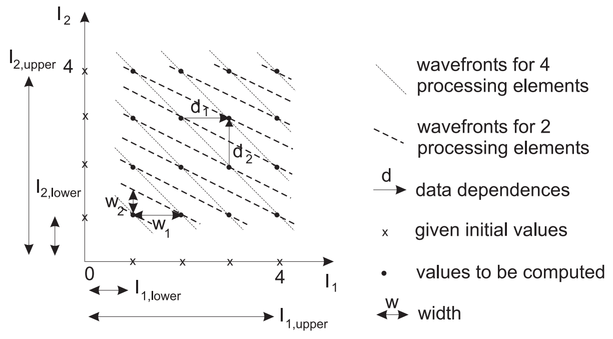


Fig. 5. Wavefronts for current example

we can compute n_{it} the number of iteration steps. Here, $I_{k,upper}$ and $I_{k,lower}$ $(1 \leq k \leq m)$ is the upper and the lower limit of the loop L_k. The parameter w_k is the width of the wavefront in direction of the I_k axis and depends on the chosen wavefronts. Figure 5 indicates that in the case of the wavefronts with 4 processing elements 7 iteration steps and in the other case 10 iteration steps are necessary. Since the index functions of the lower and the upper loop limits are affine-linear the index space is always of a trapezoidal form as shown in Figure 6, the same holds for higher dimensions. Because of this form the wavefront with the highest number of operations to be executed in this iteration step passes either through the corner c_1 or c_2. All the wavefronts left from the one passing through c_2 and right from c_1 have less operations to be computed and those between the wavefronts through c_1 and c_2 have less, too [13]. Before executing the transformation it is thereby possible to determine only on the basis of the data dependences and the wavefronts the number of iteration steps and processing elements, and the optimal wavefronts can be selected. After this selection the transformation matrix is set up as described in [13].

Another method to transform the code to the optimal degree of parallelism is to parallelize it first and then to use tiling [14]. Tiling maps a n-deep loop nest LN_n to a m-deep loop nest LN_m^{ti} where $n+1 \leq m \leq 2n$. By this mapping at least one of the loops is decomposed in the following form:

```
$L_{ii}$: **for** ii in p to q step r **loop**
    $L_i$: **for** i in ii to min(ii+r-1,q) **loop**
```

L_{ii} is put as outermost loop of the loop nest. An example is shown in Figure 7.

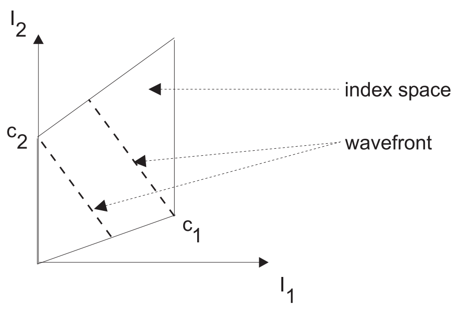
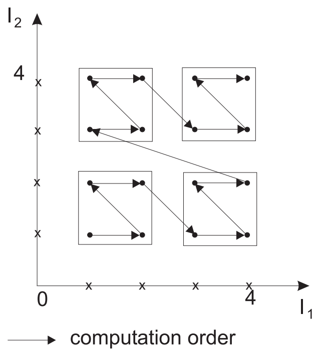


Fig. 6. Maximal number of operations



```
Lii1: for ii1 in 1 to 4 step 2 loop
   Lii2: for ii2 in 1 to 4 step 2 loop
      Li1: for i1 in ii1 to min(ii1+1,4) loop
         Li2: for i2 in ii2 to min(ii2+1,4) loop
                  ...
         end loop;
       end loop;
    end loop;
end loop;
```

Fig. 7. Tiling

With both transformation strategies it is possible to parallelize the above given code. The disadvantage of the second method is that we may need more iterations where some of them may be empty (see Figure 8). The empty iterations need unused computation

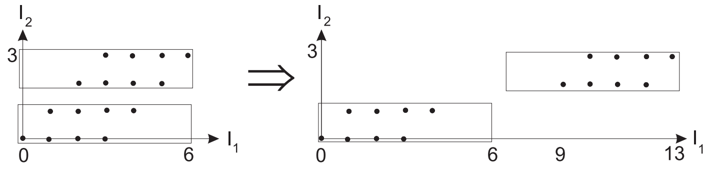


Fig. 8. Iterations without operations

time and prevent a further reduction of the voltage. Within the second strategy the transformation to the higher degree allows to select the wavefronts in such a way that the number of memory accesses is minimized and the tiling enables a reduction of the degree of parallelism. Thus, with this transformation method the designer is able to optimize the memory accesses. In order to find the optimal wavefronts in the first step the following cases of data dependences have to be considered:

I. The data dependences are given in such a way that there is no subspace of the index space which is independent of its complement. A subspace is called independent of its complement if there is no data reference from the subspace to the complement and vice versa (cf. to the current example in Figure 5).
II. The dependences are defined such that there is at least one subspace of the index space which is completely independent of its complement (see Figure 9). Cases (a) and (b) are equivalent because (b) can be transfered to (a) by loop permutation.

If the index space can be separated in independent subspaces then the given loop nest has to be split into two independent nests. Each of these loop nests corresponds to case (I). In order to select the optimal wavefronts only one loop nest has to be taken into consideration because the data dependences are uniform over the whole index space. The aim is to use those wavefronts which unify as many data references as possible and if there are two wavefronts with the same number of data references the one with the higher degree of parallelism should be selected (see Figure 10). If the degree of parallelism of the chosen wavefronts is too high the loop nest can be tiled. The results of both transformation strategies for the current example can be seen in Figure 11, in the first case we need 23 and in the second case 18 accesses to the memory, but after the transformation 1 it takes 10 and after the transformation 2 it takes 14 iterations to compute the array values because it is not possible to avoid with the second approach the empty iterations. Both strategies reduce simultaneously the power consumed by the processing and the memory elements, but we propose to apply the transformation strategy 1 in case of computational intensive applications in order to reduce the supply voltage as far as possible and to use the other method in case of memory intensive applications in order to reduce the number of memory accesses.

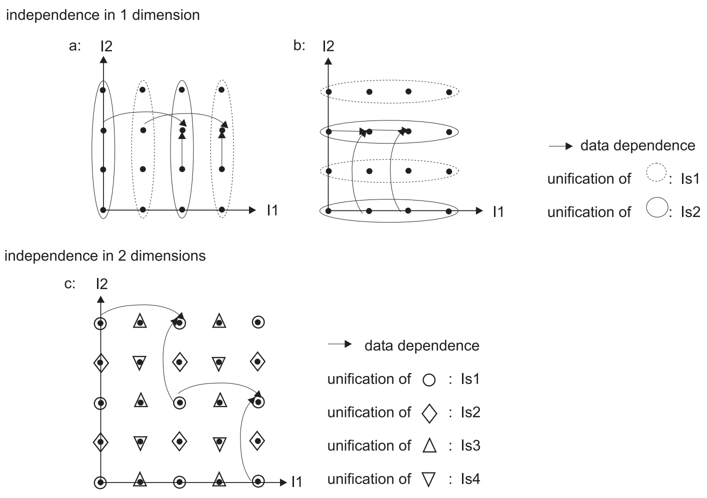


Fig. 9. Independent subspaces

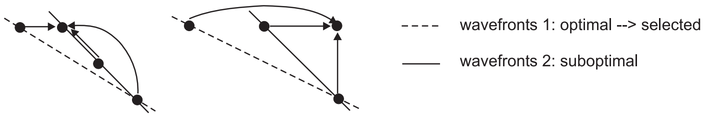


Fig. 10. Optimal wavefronts

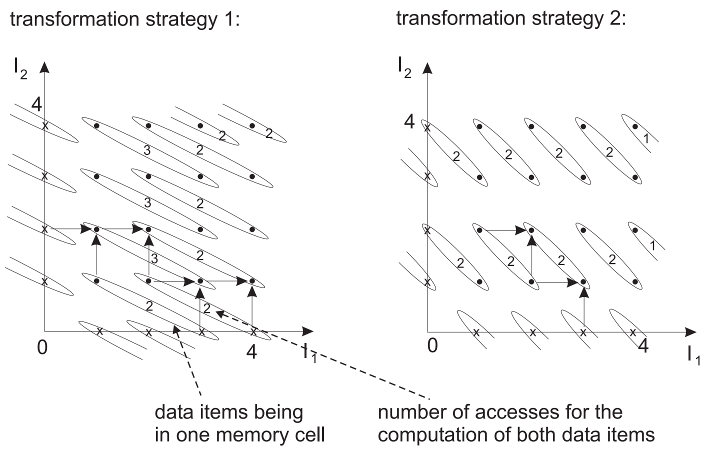

transformation strategy 1:
transformation strategy 2:
I2
4
0
4
I1
data items being in one memory cell
number of accesses for the computation of both data items

Fig. 11. Memory accesses

5 Formal Proof of Correctness

The proofs in this section were performed using the industrial-strength verification tool PVS [9]. At the end of this section we will conclude with some remarks concerning the suitability of PVS for our verification task.

5.1 Modeling Sequential and Parallel Systems

A digital system is often modeled as a (nondeterministic) automaton, i.e., a tuple $A = (S, I, T)$ consisting of a non-empty state set S, a non-empty set of starting states $I \subseteq S$ and a state transition relation $T \subseteq S \times S$. We will use the notation of guarded commands to formulate such automata. In this notation, the set of states is represented by declaring a set of variables. The starting state is defined by assigning an initial value to all variables. The state transition relation is given by a set of guarded commands. A guarded command is written as:

$$\langle \text{guard} \rangle \rightarrow \langle \text{command} \rangle$$

The guard is a boolean predicate over the system state and the command is an assignment to the variables. A guarded command is *enabled* in a state s if its guard evaluates to true in s.

It is assumed that A starts its operation in a state $s_0 \in I$. It then non-deterministically selects an enabled guard and executes the associated command, resulting in a state s_1. This procedure is repeated resulting in a sequence $\sigma = s_0, s_1, s_2, \ldots$ which is called an *execution* or *trace* of A. Due to possible non-determinism, a system can produce many different traces. The system model assumes that state transitions are atomic.

Non-determinism is the central means of modeling concurrency. The set of guarded commands is partitioned into n sets representing the programs of n concurrent processes. If two actions of different processes are enabled in a state s, both actions are equally possible to occur and can be seen as *parallel*. Figure 12 shows two parallel instances of a process which continuously keeps switching between the states 1 and 2 forever. So it is also possible to model (partly) deterministic systems in this formalism.

```
program $p$
variables: $x \in \{1, 2\}$ initially $1$
     $y \in \{1, 2\}$ initially $1$
guarded commands:
$p_1 : x = 1 \rightarrow x := 2$
$p_1 : x = 2 \rightarrow x := 1$
$p_2 : y = 1 \rightarrow y := 2$
$p_2 : y = 2 \rightarrow y := 1$
```

Fig. 12. An example concurrent program.

5.2 Properties and Correctness

Formally, a *property* P is a set of executions. By choosing which executions are in P and which ones are not, it is possible to specify certain desired properties like, for example, P_{me} denoting *mutual exclusion*. Formally, the mutual exclusion property P_{me} is the union of all executions that consist only of states where no two processes are in their critical sections at the same time.

The semantics of a system are the set of all its executions and so a system A defines a property in itself. If a system A exhibits a trace which is not an element of some property P, we say that A *violates* P. For example, if it is possible to construct a trace σ of A where at one point two processes are in their critical sections at the same time, then A violates mutual exclusion. Hence, a system A *satisfies* a property P iff (if and only if)

$$\text{“Semantics of } A\text{”} \subseteq P$$

In this case we will say that A is *correct* regarding P. Note that A must not exhibit *every* trace in P. This reflects the fact that it is possible to build many different systems which satisfy a property like for example mutual exclusion.

Since it is quite hard to define a property P by enumerating all its traces, we use a special formalism to do this. For concurrent systems the most general formalism for this task is temporal logic [7]. We restrict here our attention to properties which can be expressed in the form “always φ”, where φ is a predicate over the system state. In temporal logic, this is written as $\Box\varphi$ and means the set of all possible traces consisting only of states where φ holds.

Verification means to show that a given system A satisfies a property P. This can be done by using standard approaches of theoretical computer science. To show that a program A satisfies a property of type $\Box\varphi$ it suffices to show the following two proof obligations:

(B) φ holds in the initial state of A, and
(S) for all possible guarded commands of A: Assuming φ holds in a state s, then φ also holds in the state resulting from executing the guarded command in s.

Note that the principle behind this rule is that of induction. The first step is used to prove the base case and the second step is used to prove the induction case.

5.3 The Basic Building Block of the Transformations

The basic method applied by the transformation is that of *correctness-preserving parallelization*. In our terms it means the following: A program must never make “bad” computations. For example, take the loop construct from Figure 4: The array element $A(x,y)$ can be computed only if $A(x-1,y)$ and $A(x,y-1)$ have been computed already. More precisely, the sequence of computing the array elements $A(x,y)$ must respect the dependency graph of the loop construct. Note that this perspective allows to neglect the actual values present in A.

We will continue to use the example from Figure 4, but we will simplify it by reducing the loop bounds from 4 to 2. While retaining the potential of parallelization, this simplification makes it easier both to handle the example in the proof system PVS and to present it in this paper. With the reduced loop bounds, we can formalize the dependency graph as follows: Let c denote a boolean 3×3 array where $c(x, y) = \mathsf{true}$ means that array element (x, y) has been computed already. We let the indexes range between 0 and 2 and assume that all elements where either x or y is 0 are border elements (e.g., $c(0, 0)$, $c(0, 1)$ etc.). All border elements are initialized to true while all other values are false. The property which both the sequential and the transformed program must satisfy can then be defined using the following state predicate:

$$\varphi \equiv \forall x, y \in \{1, 2\} : c(x, y) \Rightarrow c(x - 1, y) \wedge c(x, y - 1)$$

The initial state of our programs is described by the following state predicate:

$$\gamma \equiv \forall x, y \in \{0, 1, 2\} : c(x, y) \Leftrightarrow x = 0 \vee y = 0$$

It is relatively easy to see that γ implies φ. The idea behind the proof in PVS is that all inner elements of c are initially false and so the antecedent of the implication in φ never holds. Thus, the implication in φ is true in all cases. Hence, γ implies φ.

5.4 Proving the Correctness

We performed the correctness proof for two programs (see Figures 13 and 14): A is a sequential one which calculates the elements according to the loop code in Figure 4. Program B is the parallelized version which allows the concurrent computations of elements along the wavefronts explained in Section 4.

```
program $A$
variables: $l \in \{1, \ldots, 5\}$ initially 1
      $c[0..4, 0..4]$ initially satisfies $\gamma$
guarded commands:
$l = 1 \rightarrow c(1, 1) := \mathsf{true};\ l := 2$
$l = 2 \rightarrow c(1, 2) := \mathsf{true};\ l := 3$
$l = 3 \rightarrow c(2, 1) := \mathsf{true};\ l := 4$
$l = 4 \rightarrow c(2, 2) := \mathsf{true};\ l := 5$
```

Fig. 13. The sequential program.

Note that in program B, there are two guarded commands with the guard $l = 2$. This is where the potential concurrency is encoded.

The initial state of both programs is the same ($l = 1$ and c satisfies γ) and since γ implies φ (as shown above), this initial state also satisfies φ. This is sufficient to show the proof obligation (B) for the base case.

Consider now proof obligation (S) and program A. A common technique for showing that A satisfies (S) uses a special state predicate called an *invariant*. An invariant is a state predicate α which implies φ and remains true throughout the execution of the

```
program B
variables: $l \in \{1, \ldots, 6\}$ initially 1
    $c[0..4, 0..4]$ initially satisfies $\gamma$
guarded commands:
$l = 1 \rightarrow c(1,1) := \mathsf{true};\ l := 2$
$l = 2 \rightarrow c(1,2) := \mathsf{true};\ l := 3$
$l = 2 \rightarrow c(2,1) := \mathsf{true};\ l := 4$
$l = 3 \rightarrow c(1,2) := \mathsf{true};\ l := 5$
$l = 4 \rightarrow c(2,1) := \mathsf{true};\ l := 5$
$l = 5 \rightarrow c(2,2) := \mathsf{true};\ l := 6$
```

Fig. 14. The parallelized program.

program. The invariant has a special form which facilitates proving (S); thus, finding the invariant is not always easy since it must bear in itself the "idea" of the correctness proof. In our case, it is not so difficult to state the invariant. It is as follows:

$$\begin{aligned}\alpha_A \equiv \varphi &\wedge (l = 2 \Rightarrow c(1,1))\\ &\wedge (l = 3 \Rightarrow c(1,1) \wedge c(1,2))\\ &\wedge (l = 4 \Rightarrow c(1,1) \wedge c(1,2) \wedge c(2,1))\\ &\wedge (l = 5 \Rightarrow c(1,1) \wedge c(1,2) \wedge c(2,1) \wedge c(2,2))\end{aligned}$$

Clearly, the invariant implies φ. However, it additionally captures the "filling" of the array by the program by formalizing the correspondence between l and c in a sequence of implications.

The proof in PVS is performed by showing the implication of proof obligation (S) for every guarded command of A. The individual proofs are performed by case analysis and by instantiating the general implication in φ with concrete values to prove the different cases.

The invariant for proving that B satisfies γ is similar to α_A. It is:

$$\begin{aligned}\alpha_B \equiv \varphi &\wedge (l = 2 \Rightarrow c(1,1))\\ &\wedge (l = 3 \Rightarrow c(1,1) \wedge c(2,1))\\ &\wedge (l = 4 \Rightarrow c(1,1) \wedge c(1,2))\\ &\wedge (l = 5 \Rightarrow c(1,1) \wedge c(1,2) \wedge c(2,1))\\ &\wedge (l = 6 \Rightarrow c(1,1) \wedge c(1,2) \wedge c(2,1) \wedge c(2,2))\end{aligned}$$

The proof uses the same techniques as the proof for program A.

Overall, we have shown that both programs maintain the invariant. Since the invariant implies φ both programs also maintain φ. This shows the second proof obligation (S) and thus concludes the correctness proofs: A and B both satisfy $\Box\varphi$.

5.5 Remarks on Using PVS

The verification of the example programs have been our first real experiences with PVS. Overall, they turned out to be a good exercise to get started with the tool. It took us about one week to formulate the problem in the PVS specification language and verify it using the prover. Although the fixed state space of our problem suggests to use the model checking features of PVS, we used the theorem prover. After getting used with the system, proving the individual theorems required only half a dozen user interactions.

6 Summary

In this paper two parallelizing transformation strategies have been proposed which reduce both, the energy dissipated in the processing elements by the reduction of the supply voltage and the energy consumed in the memory by minimizing the number of accesses. By these transformation strategies it has become possible to reduce the energy dissipation in the processing elements and the memory up to 50% and it was possible to give a formal proof that the parallelizing transformation preserves the correctness.

References

1. U. Banerjee. *Loop Transformations for Restructuring Compilers: The Foundations.* Kluwer Academic Publishers, 1993.
2. U. Banerjee. *Loop Transformations for Restructuring Compilers: Loop Parallelization.* Kluwer Academic Publishers, 1994.
3. J. Becker. *A Partitioning Compiler for Computers with Xputer-based Accelerators.* PhD thesis, Kaiserslautern University, 1997.
4. F. Catthoor, S. Wuytack, E. De Greef, F. Balasa, L. Nachtergaele, and V. Arnout. *Custom Memory Management Methodology.* Kluwer Academic Publishers, 1998.
5. A. P. Chandrakasan and R. W. Brodersen. Minimizing Power Consumption in Digital CMOS Circuits. *Proceedings of the IEEE*, 83(4):498 – 523, April 1995.
6. J.-M. Chang and M. Pedram. *Power Optimization And Synthesis At Behavioral And System Levels Using Formal Methods.* Kluwer Academic Publishers, 1999.
7. E. A. Emerson. Temporal and modal logic. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, chapter 16, pages 997–1072. Elsevier, 1990.
8. S. Y. Kung. *VLSI Array Processors.* Prentice Hall Information and System Sciences Series, 1988.
9. S. Owre, S. Rajan, J. Rushby, N. Shankar, and M. Srivas. PVS: Combining specification, proof checking, and model checking. In R. Alur and T. A. Henzinger, editors, *Computer-Aided Verification, CAV '96*, number 1102 in Lecture Notes in Computer Science, pages 411–414, New Brunswick, NJ, July/August 1996. Springer-Verlag.
10. P. R. Panda, N. Dutt, and A. Nicolau. *Memory Issues in Embedded Systems-On-Chip.* Kluwer Academic Publishers, 1999.
11. J. M. Rabaey and M. Pedram. *Low Power Design Methodologies.* Kluwer Academic Publishers, 1996.
12. A. Raghunathan, N. K. Jha, and S. Dey. *High-Level Power Analysis and Optimization.* Kluwer Academic Publishers, 1998.

13. M. Theisen, J. Becker, M. Glesner, and T. Caohuu. Parallel hardware compilation in complex hardware / software systems based on high-level code transformations. In *ARCS'99: 15. GI/ITG - Fachtagung: Architektur von Rechensystemen*, October 1999.
14. M. E. Wolf. *Improving Locality and Parallelism in Nested Loops.* PhD thesis, Stanford University, 1992.