RESOURCE ALLOCATION FOR A MOBILE APPLICATION ORIENTED ARCHITECTURE

Yuanqing Guo Gerard J.M. Smit
Faculty of EEMCS, University of Twente
the Netherlands
E-mail: y.guo@utwente.nl

Abstract

A Montium is a coarse-grained reconfigurable architecture designed by the CADTES group of the University of Twente for mobile applications. This paper presents a resource allocation method to allocate variables to storage places and to schedule data movements for the Montium. The resource allocation method exploits locality of reference of the Montium architecture as well as its parallelism.

1 Introduction

A key challenge of mobile computing is that many attributes of the environment vary dynamically. Mobile devices must be flexible enough to be able to adapt to a new environment. Since more and more applications are added to mobile computing, speed is another important issue for mobile computing. Furthermore a handhold device should be able to find its place in a pocket. This implies an ultra-low energy consumption. To summarize above, mobile computing should be flexible, with high performance and energy efficient.

A Montium [7] is a coarse-grained reconfigurable architecture designed for mobile applications. Figure 1 depicts a single Montium processor tile. Each Montium processor tile consists of five identical Processing Parts (PPs), which share a control unit. An individual PP contains an arithmetic and logic unit (ALU), four input register banks named Ra, Rb, Rc, Rd and two memories called MEM1 and MEM2. Each register bank consists of four registers. Each memory has 512 entries. A crossbar-switch makes flexible routing between the ALUs, registers and memories possible. The crossbar enables an ALU to write back their result to any register or memory within a tile.

Currently a compiler is being designed for the Montium by the CADTES group at the University of Twente. In this paper, we present the resource allocation method used by the Montium compiler. The resource allocation includes allocating variables to storage places and scheduling data movements. Variables are evenly allocated to memories to maximize the parallelism of data movements and consequently to prevent the writing/reading bottleneck of memories. Meanwhile variables are allocated to the memories which are local to the ALUs which access them. Data movements are scheduled such that as few as possible clock cycles are needed to run an application program.

2 Related work

The storage allocation problem has been studied for a long time. The first paper on this subject appears to be Lavrov's 1961 paper on minimizing memory usage [6]. Ershov solved storage allocation problems by building an interference graph and using a packing algorithms on it [5]. Some papers focus on register allocation. Chaitin et al. first used graph coloring as a paradigm for register allocation and assignment in a compiler [1]. Briggs et al. describe a variation of the coloring heuristic that increases the number of live ranges that can be colored [2][3]. Chow and Hennessy describe a priority-based coloring scheme [4].

3 Resource Allocation

For allocation, we assume every operation of an application program has been assigned an ALU and the relative executing order of ALUs has been determined. The decisions that should be made during the allocation phase include choosing proper storage places (memories or registers) for each value and scheduling data movements under the constraints of resource limitations

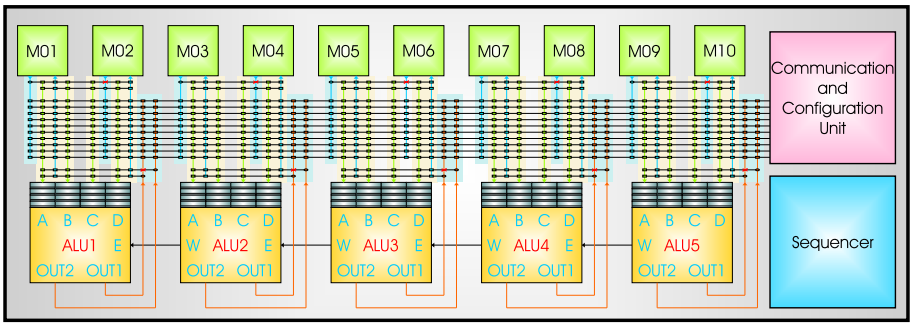

Figure 1: Montium processor tile

```
float C1, C2, C3, C4, C5;
float T0 = T1 = T2 = T3 = T4 = 0;
float input[512]; //this is the input array.
float output[512]; //this is the output array.
Fir5(){
 for(int i=0; i<512; i++){

    T4 = input[i]×C4 + T3;    //(clu0)
    T3 = input[i]×C3 + T2;    //(clu1)
    T2 = input[i]×C2 + T1;    //(clu2)
    T1 = input[i]×C1 + T0;    //(clu3)
    T0 = input[i]×C0 ;        //(clu4)
    output[i] = T4;
 }
}
```

Figure 2: The allocation input for FIR5 program

(such as reading/writting ports of memories, number of registers, crossbar, size etc).

Given an example of FIR5 filter (See Figure 2). The mappings of operations to ALUs are as follows:

operation or variable	clu0	clu1	clu2	clu3	clu4
destination	ALU1	ALU2	ALU3	ALU4	ALU5

3.1 Allocate external variables to storage places

To run an application program on a Montium tile, external input variables (except for streaming inputs) should be stored in memories after the configuration stage and after finishing the program, external outputs should be saved in memories (except for streaming outputs). Two aspects need to be taken into account when choosing a memory for a variable or an array: (1) To have memories occupied evenly; (2) To exploit the locality of reference.

The first aspect is to maximize the parallelism. Since only one value can be read from or written to a memory at each clock cycle, and when too many values are staying in the same memory, the execution speed might be reduced due to the reading and writing bottleneck of the memory.

The locality of reference principle is applied in several levels of the Montium structure: local memories for each PP, local registers for each ALU. One input value might be used by ALUs of different PPs, so it is not always possible to put the value into a memory that is local to all ALUs which use it. However, the PP that uses the value most frequently can be found. When the input is located in a memory of that found PP, the frequency of using global buses of the crossbar will be minimized. The algorithm for allocating external variables and arrays is:

foreach variable Var{
  Compute how many times it is used by each ALU i and put it to TimesUsedByALU[i];
  Allocate Var to the memory which has the largest value of $f(k)$;
}

The definition of $f(k)$ should satisfy two criterions:

1 $f(k)$ is an increasing function of TimesUsedByALU[$\lceil k/2 \rceil$], where $\lceil d \rceil$ represents the smallest integer larger than d. $\lceil k/2 \rceil$ is thus the index of the PP to which memory Mk belongs.

2 $f(k)$ also increases as the number of empty entries of memory Mk.Empty increases, where Mk.Empty is

the number of empty entries in memory Mk.

The computed TimesUsedByALU[i] for each arrays and variables for the example of Figure. 2 are listed in Table 1.

Table 1: Values of TimesUsedByALU[i]

	TimesUsedByALU[i]				
Name	$i=1$	$i=2$	$i=3$	$i=4$	$i=5$
Input	512	512	512	512	512
Output	512	0	0	0	0
C0	0	0	0	0	512
C1	0	0	0	512	0
C2	0	0	512	0	0
C3	0	512	0	0	0
C4	512	0	0	0	0
T0	0	0	0	512	512
T1	0	0	512	512	0
T2	0	512	512	0	0
T3	512	512	0	0	0
T4	512	0	0	0	0

The allocation procedure starts from the array Input, since the $f(k)$ has the same value for every memory Mk, there is no preference in choosing a particular memory. The algorithm picks one at random: Input is allocated in M1. Array Output is only accessed by ALU1, so the algorithm tries to put Output to a memory of PP1. Since M1 is occupied, $f(2) > f(1)$, M2 is the best choice for array Output. Next the variables are allocated. The allocation is done automatically and the result is shown in Figure 3. We can see that C0, C1, C2, C3, T0, T1, T2 and T3 are allocated locally to the ALUs which needs them. C4, T4 are used by ALU1, but they are not put into the memories of PP1. This is because M1 and M2 are already occupied.

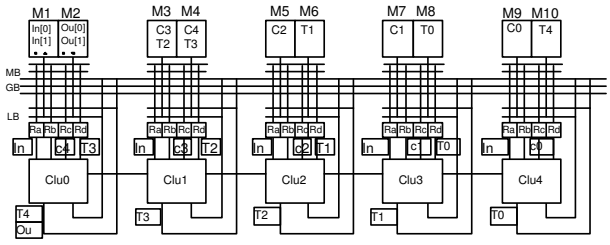

Figure 3: Allocation of variables

3.2 Define data movements

There are two purposes of data movements: (1) Arrange the resources (crossbar, address generators, etc) such that the outputs of the ALUs are saved in the proper registers and/or memories; (2) Arrange the resources such that the inputs of ALUs are in the proper register for the next cluster that will execute on that ALU. For the physical implementation, loading a copy of a value from a memory to a register is called a *fetch* (FE) and saving an output to a memory is called a *store* (ST). An output can also be saved to a register, we call it consignment (CS). A value can also be copied to from one memory to another memory. We call it shift (SH).

3.3 Combine data movements

In many cases especially within the loop body, some data movements can be combined. This combination will decrease the pressure on the crossbar and memory ports. We will concentrate on loop bodies to discuss the situations of combinations of movements.

Combine FEs: In a loop body, some variables are only FEed. Thus the value is always the same during the entire loop. In Figure 2, parameters C0, C1, C2, C3 and C4 are such kind of variables. In this case, instead of defining a FE movement for every iteration, we can move these FEs for all iterations to the front of the loop. Thus the FE movement only has to be done once. The disadvantage of this combining is that the register which keeps the value cannot be used by others during the execution of the whole loop. Thus this combining can only be done when there is a free register.

Combine STs with FEs or SHs: If a value is first stored to a variable and then fetched from that variable immediately, then the consumers of the fetched value can operate directly on the first producer of the first value. Take $T0$ in Figure 2 as an example, T0 saved in the previous iteration will be used at the next iteration. The value of T0 can be saved to Rd of ALU4 directly.

Combine STs: In a loop body, if there is a ST for a variable at every iteration, then only the ST of the last iteration is needed. For instance, in Figure 2 we can use the ST for T0 of last iteration to substitute the STs for T0 of all iterations.

3.4 Break a data movement

Every data movement can be broken up into several movements. A SH operation from PP1 memory 1 to PP2 memory 3 can be changed into two steps: first the value is moved to, for instance, PP1 memory 2, and then it is moved to PP2 memory 3. Breaking movements gives more flexibility to the compiler. It, however, also adds more pressure to crossbar and memory/register ports. To decrease the complexity of the compiler, we do not break other movements except for STs and for CSs. Since STs and CSs save outputs of ALUs to memories and registers, the clock cycle to save an output must be the clock cycle following the one in which the ALU is executed. At this specific clock cycle, if a value corresponding to the CS or ST cannot be saved, the value will be lost. This case unfortunately can happen when the reading/writing ports of destination memories/registers are occupied. The solution is to save the output to a temporary memory whose reading/writing port is not busy at that clock cycle, and then copy it to the correct destination later when its reading/writing port is free. Thus a CS is broken into a ST and a FE and a ST is broken into a ST and a SH. A Montium tile has ten global busses and the maximal number of outputs produced by ALUs within one clock cycle is also 10. If we give the crossbar usage for saving outputs priority over loading inputs, then the crossbar is always enough for saving outputs. This is shown in section 3.6. For this reason, the crossbar is not an important factor when breaking a movement.

3.5 Spill values from registers

Each register bank in the Montium has four entries. Registers are supposed to be enough when we define data movements. If at some moment, the number of entries of a specific register bank is larger than 4, one or more values should be spilled, i.e., saved at a memory and fetched to register before being used. The traditional graph coloring method of choosing the proper values can be found in [2][1]. We do not address it here again. For the Montium the graph coloring method has to be used on each register bank (totally 20). Once a value representing a movement CS is chosen to spill, the CS is broken up into a ST and a FE as shown in section 3.4. And the FE operation is done just before using it. If a value of FE is chosen to be spilled, then we only need to limit the moment of the FE and let it be done just before being used.

3.6 Schedule data movements

Storing an ALU result must be done in the clock cycle immediately following the clock cycle in which the output is computed. If an output of a cluster is not moved to registers or memories immediately after generated by ALUs, it will be lost. For this reason, in each clock cycle, storing outputs of the previous clock cycle takes priority over using the resources. Preparing an input should be done at least one clock cycle before it is used. If the inputs are not well prepared before the execution of an ALU, one or more extra clock cycles need to be inserted to do so. However, this will decrease the runtime of the algorithm.

The procedure of data movement scheduling can be presented by the code in Figure 4.

```
ScheduleDataMovements(){
  foreach CS or ST{
    if the port of its destination is free{
      Allocate resources (storage places, reading/writing ports
and crossbar) for it;
    }
    else{
      Break the CS to ST and FE or break the ST to ST and SH
according to section 3.4;
      Allocate resources for the new ST.
    }
  }
  foreach FE or SH{
    if the ports of its destination and source are free at any clock
cycle
    before the clock when the value should be ready{
      Allocate resources (storage places, reading/writing ports
and crossbar) for it;
    }
    else{
      Insert a new empty clock cycle before the clock which use
the value.
      Allocate resources at the new inserted clock cycle.
    }
  }
```

Figure 4: Pseudocode for scheduling movements

4 Simulation and Result

We simulated our the resource allocation algorithm. External inputs and outputs are allocated as Figure 3. The defined movements are listed in Table 2.

Then some movements can be combined.

1. For $c4$, $c3$, $c2$, $c1$ and $c0$, the FEs of all iterations can be combined and moved to the begin of the loop.
2. For $T4$, the STs of all. iteration can be combined and moved to the end of the loop
3. Combine the ST and FE of $T3$ a CS from PP3.O1 to

Table 2: Defined movements for FIR5

Index	Name	Move		
		From	To	Type
1	$c4$	M4	PP1.Rc	FE
2	$In[k]$	M1	PP1.Ra	FE
3	$T3$	M4	PP1.Rd	FE
4	$c3$	M3	PP2.Rc	FE
5	$In[k]$	M1	PP2.Ra	FE
6	$T2$	M3	PP2.Rd	FE
7	$c2$	M5	PP3.Rc	FE
8	$In[k]$	M1	PP3.Ra	FE
9	$T1$	M6	PP3.Rd	FE
10	$c1$	M7	PP4.Rc	FE
11	$In[k]$	M1	PP4.Ra	FE
12	$T0$	M8	PP4.Rd	FE
13	$c0$	M9	PP5.Rc	FE
14	$In[k]$	M1	PP5.Ra	FE
15	$Ou[k]$	PP1.O1	M2	ST
16	$T4$	PP1.O1	M10	ST
17	$T3$	PP2.O1	M4	ST
18	$T2$	PP3.O1	M3	ST
19	$T1$	PP4.O1	M6	ST
20	$T0$	PP5.O1	M8	ST

PP2.Rd. The similar combination is done for STs and FEs of $T0$, $T1$, $T2$.

After the combination, the occupation of register banks can be summarized as table 3. The first group of

Table 3: Variables in registers

PP1				PP2			
Ra	Rb	Rc	Rd	Ra	Rb	Rc	Rd
$In[k]$		$c4$	$T3$	$In[k]$		$c3$	$T2$
PP3				PP4			
Ra	Rb	Rc	Rd	Ra	Rb	Rc	Rd
$In[k]$		$c2$	$T1$	$In[k]$		$c1$	$T0$

PP5			
Ra	Rb	Rc	Rd
$In[k]$		$c0$	

movements that are scheduled are: ST($Ou[k]$), CS($T3$), CS($T2$), CS($T1$), CS($T0$). These movements are scheduled within each loop body: i.e., a movement should be executed once for each iteration. Then FE($In[k]$)s are scheduled; FE($c4$), FE($c3$), FE($c2$), FE($c1$) and FE($c0$) are done before the loop; ST($T4$) is executed after the loop.

Using our resource allocation, each iteration of the loop body of FIR5 can be executed within one clock cycle except the prelude and postlude. This is the same as our hand-made result.

5 Conclusion

This paper presents a heuristic resource allocation method to exploit locality of reference as well as providing high performance. The locality of reference principle of difference level is embodied: Allocate external variables or arrays local to the ALUs which need them; save outputs of ALUs to registers directly (combine ST and FE to CS); keep constants in proper registers (Combine FEs and move it to the front of loop body). Parallelism is materialized by: evenly occupation of memories, scheduling of data movements. The simulation results show the effect of the resource allocation algorithm.

The goal of storage allocation (including register allocation) is to minimize the number of loads and stores that must be executed under the constraints of the size of a given register file. Due to the distributed register files, for the Montium, the scheduling of data movements has a strong influence on the performance of system. Therefore, the resource allocation for the Montium should consider not only storage allocation but also data movements.

References

[1] Gregory J. Chaitin,, "Register allocation and spilling via graph coloring", *SIGPLAN Notices 17(6)*, 98-105, 1982. In *Proceedings of the ACM SIGPLAN '82 Symposium on compiler Construction.*

[2] Preston Briggs, *Register Allocation via Graph Coloring.* PhD thesis, Rice University, April, 1992.

[3] Preston Briggs, Keith D. Cooper, and Linda Torczon, "Improvements to Graph Coloring Register Allocation", In *ACM Transactions on Programming Langruages and Systems*, Vol. 16, No. 3, May 1994, pp. 428-455.

[4] F.C. Chow, and J.L. Hennessy, " The priority-based coloring approach to register allocation", *ACM Transactions on Programming Languages and Systems 12*, 4(Oct. 1990), 501-536.

[5] A.P. Ershov, "Reduction of the problem of memory allocation in programming to the problem of coloring the vertices of graphs", *Doklady Akademii Nauk S.S.S.R. 142, 4(1962)*, English translation in *Soviet Mathematics 3:163-165*, 1962.

[6] S.S. Lavrov "Store economy in closed operator schemes". *Journal of Computational Mathematics and Mathematical Physics 1*, 4(1961), 687-701.

[7] Paul M. Heysters, Gerard J.M. Smit, E. Molenkamp: "A Flexible and Energy-Efficient Coarse-Grained Reconfigurable Architecture for Mobile Systems", *The Journal of Supercomputing*, volume 26, number 3, Kluwer Academic Publishers, Boston, U.S.A., November 2003, ISSN 0920-8542.