A Compiler with Scheduling for a Specialized Synchronous Multiprocessor System

Petr Kroha
Technical University Praque, Department of Computers,
FEL-CVUT, Karlovo nam.13, 121 35 Praha 2, Czechoslovakia.

Peter Fritzson[*]
E-mail: paf@ida.liu.se
Department of Computer and Information Science
Linköping University, S-581 83 Linköping, Sweden

Abstract: This paper presents an algorithm for scheduling parallel activities in a specialized synchronous multiprocessor system. The algorithm is being implemented as a part of a cross-compiler for an extended parallel Single Instruction Computer (SIC). A SIC machine may contain multiple arithmetic processors, each associated with certain addresses in the address space.

The scheduling cross-compiler initially derives a schedule including information about the number and types of processors necessary for the highest possible degree of parallelism for the code in each basic block. If too few arithmetic processors are available, a schedule for a smaller number of processors can be generated. Code generation and scheduling is presented for a one page program example in Pascal. For this example, a speedup of a factor of seven was obtained for the multiprocessor system, compared to the Intel 80286 processor, and assuming the same clock cycle time.

Introduction

Besides the classical computer architectures there are many attempts at new architectures.. There are controversies concerning whether the instruction set should be very simple or very complex. We do not settle this controversy here; there can hardly be a simple answer. Several successful attempts belong to the RISC architecture described in many papers and books, e.g. [Pat82], [Mil86], [BlaKro87] which uses a reduced set of instructions (about 40). It has been demonstrated, e.g. by [Kro88], that the gain in performance by far overweighs the loss of the possibility of hand coding in assembly language, and somewhat larger code size. RISC machines such as the Sun SPARC [Nam88] have a fascinating performance.

However, there is an extreme possibility called the single instruction machine with only one instruction. This idea was first published by Lipovski,G.J. [Lip76]. Such a machine formally executes only one instruction, and thus has no instruction set, no opcode and no instruction decoding. Instead, different operations are executed by Arithmetic Units (AMUs) associated with special addresses in the address space. Hardware realizations and applications are described in [TabLip80], [AzTab80], [AzTab83].

The key idea is that a Central Move Unit (CMU) delivers values to Arithmetic Units which each are specialized in one function. The addressing mode is decoded by the CMU so that the AMUs only need to handle execution.

The main ideas of the SIC machine have been summarized in [Tab87]. The described SIC

*. This research has been supported by the Swedish National Board for Technical Development (STU).

machine uses a very modest set of AMUs. There was no attention devoted to the SIC machine from the software point of view. This was probably the reason that only such AMUs were considered whose execution time are shorter than that of a MOVE operation. From the hardware point of view this is an advantage because the AMUs can work without starting and stopping, and the machine need not consider the synchronization problem. However, that approach has several disadvantages:

- The set of operations which are faster than a MOVE operation is very small.
- For control applications such a machine would be difficult to program, e.g. multiplication of real numbers using the simplest operations.
- The potential parallelism of the SIC machine cannot be used because the MOVE operation of the CMU is the slowest one.

In the report [Kro89] some properties of an extended SIC machine are described and its programming possibilities. These extensions are:

- AMUs with an execution time longer than that of the MOVE operation, creating a base for nontrivial programming.
- Two CMUs cooperating with a two-port memory. The managing of the starts and stops of the AMUs is somewhat more complex, but this opens up possibilities for simple parallel processing.

Further it was demonstrated in [Kro89] that for every algorithm a SIC machine could be built with the same computational power as any CISC machine. Examples and a comparison with an assembly program for a PDP 11 machine was given.

This opens up the possibility of a customizable compiler with scheduling, which can compile programs into specialized parallel-processing SIC machines, which have exactly the required number and kind of AMUs.

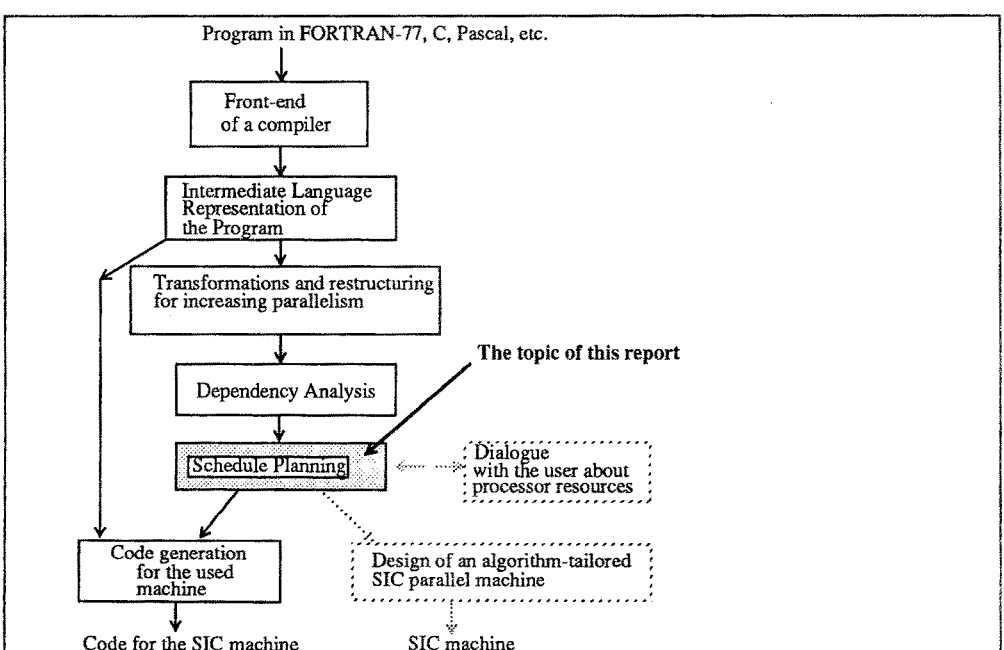

Figure 1. A schematic diagram of a compiler with scheduling

Common requirements on parallel architectures

Before starting the discussion on the usage of SIC machines for the purposes of parallel computing we will briefly overview some common problems concerning machines for parallel processing.

Problems of memory

The memory system is so important that a well-known researcher has called it "the von Neumann bottleneck" because of its critical role. It limits how fast input data can be delivered to a processor and how fast the results can be received from the processor.

In the presented approach two CMUs are working in parallel with a two-port memory to speed up the processing. This is, of course, still a bottleneck, but an improvement over a single-ported memory.

Problems of overhead

During the execution of a program on a parallel machine run-time overhead is incurred from activities such as scheduling, interprocessor communication and synchronization. This overhead is added to the execution time in the form of processors latencies and busy waits. As the overhead increases, the amount of parallelism that can be exploited decreases. There is a larger overhead for asynchronous multiprocessor parallel machines that for synchronous systems. Our approach concerns synchronous multiprocessor system.

Problems of transforming serial algorithms into parallel algorithms

In putting multiprocessor systems to use, a major hurdle is in writing programs for such architectures. One way to automate the production of parallel programs is to construct a compiler for a standard high-level language to produce output for a multiprocessor system. With such a compiler, existing software libraries can be mapped to a multiprocessor system with a minimum of effort. Some fraction of the library undoubtedly will exhibit negligible parallelism and will produce rather inefficient parallel implementations. These programs can be run serially.

Creating a high-quality optimizing compiler for a multiprocessor system is a formidable task. An early attempt [Kuck72] showed that there is exploitable parallelism on the order 10 to 100 in many ordinary FORTRAN programs. The next decade have produced far more sophisticated developments that have been used extensively for real applications. However, compilers for multiprocessors have lagged behind compilers for vector processors because the translation problem is more complex for multiprocessors.

To process two statements in parallel they must not contain *dependent variables* in the sense that a value is computed in one statement and used in another statement. Such dependencies can be detected through USE-DEF dataflow analysis.

The problem of scheduling

After program restructuring, a compiler should identify the parts of a program that can take advantage of the architectural characteristics of a machine.

To take full advantage of a machine architecture is a complex and difficult task. When a program has been transformed into parallel form, the next step in the translation process is to find ways to exploit the parallelism on a given machine architecture. Part of this is the *scheduling problem.*

The scheduling schema can be *static* (derived at the time of program translation by the compiler) or *dynamic* (derived at run-time by the hardware or the operating system).

Scheduling compilers (sometimes called auto-scheduling compilers) create plans for the execution of parallel programs without the involvement of an operating system for synchroniza-

tion. This allows for spreading tasks across processors during run-time in an efficient way which involves little overhead.

Scheduling methods are discussed in [Po88] and summarized:

"Extensive work has been done on the problem of static scheduling of independent tasks on parallel processors. Most of the instances of this problem have been proved to be NP-complete. These theoretical results however are of little help for practical cases."

A summary of problems when designing parallel architectures

When designing parallel machines, usually a class of algorithms, rather than a single algorithm, must be considered. The more difficult objective is to create a single architecture that is good for all the problems in a certain class.

In closing this section, we summarize by saying that the advantage of using the principle of an extended single instruction machine for synchronous parallel computation is that all synchronization and communication problems (including the memory sharing problems) are simply solved by using CMUs.

A Scheduling Algorithm

To get some idea about the possibilities of parallelism in a given algorithm, we need to know the "depend *on*" relation for variables and objects in memory, in the sense that if an object A "depends *on*" an object B then the computation of A cannot be started before the computation of B has been finished. The graph representation of this relation is denoted the dependency graph of variables. There can be a wide spectrum of shapes of such graphs.

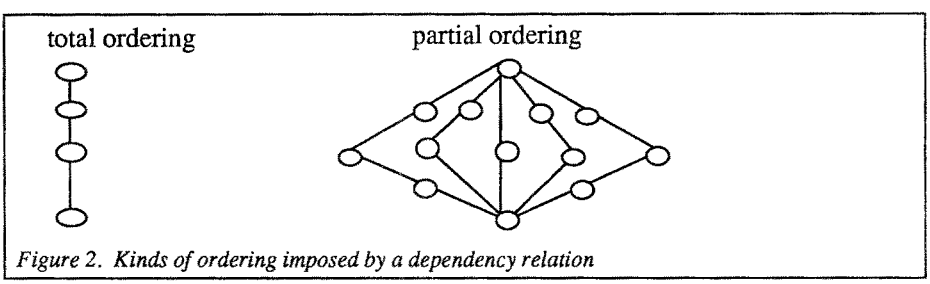

Figure 2. Kinds of ordering imposed by a dependency relation

It is obvious that the possibilities for parallel execution are different for different degrees of ordering within such graphs. There are many programs, where part of the potential parallelism is hidden and where the degree of parallelism can be increased through transformations [Fi84]. These transformations can be used by an independent phase of a compiler such as an optimizer in a classical compiler (Fig.2). Some sophisticated transformations are described in-Fi84],[Ell86].

The dependency graph of variables will be derived from the intermediate language representation of the program. It has variables as nodes and edges labeled by operations (this is easy to derive from the intermediate representation. For the purposes of scheduling we shall use this graph in its inverted form, i.e. we substitute edges by nodes and vice versa. Thus nodes will represent operations and edges will represent the variables. This new created graph we shall call a *dependency graph of AMUs*. After the compiler is ready with this dependency graph, it computes the highest number of usable AMUs of each type, e.g. the compiler informs the user that it is not necessary to use more than M arithmetic units for addition (in the algorithm tailored approach described in section 7).

We will make a difference between *virtual AMUs* and *real AMUs*. The number of virtual AMUs is straightforwardly derivable from the program in intermediate language form, where

one "new" (not used until now) AMU is supposed to be allocated for each operation. Real AMUs are AMUs which are physically available in the machine.

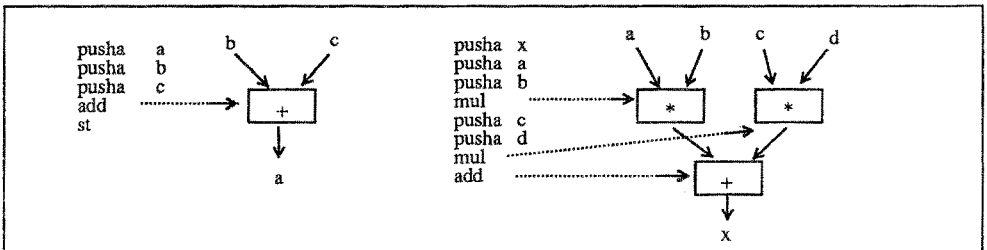

Figure 3. Intermediate language representation and dependency graph of AMUs

However, a situation can arise where the compiler will propose to use some exotic unit, because of special operations in the algorithm. These operations are given by the operators of the intermediate language. In this case the user must decide. For example, should an operator function such as *sin* really be an AMU (i.e. realized in hardware like in the 80287) or if it should be computed by a library routine call. When library calls are included, the compiler must use their source form for purposes of scheduling and substitute the call by the body of the library subroutine (only for scheduling, not for code generation), because the procedure uses the same hardware units in parallel and this must be taken into account.

When scheduling, in most cases it will be found that it is not necessary for all these virtual functional units (virtual AMU) to be mapped into real AMUs by a one to one mapping, because they cannot work fully in parallel. Therefore the virtual units (of the same type) which are used during non-overlapping time periods can be represented by one real AMU.

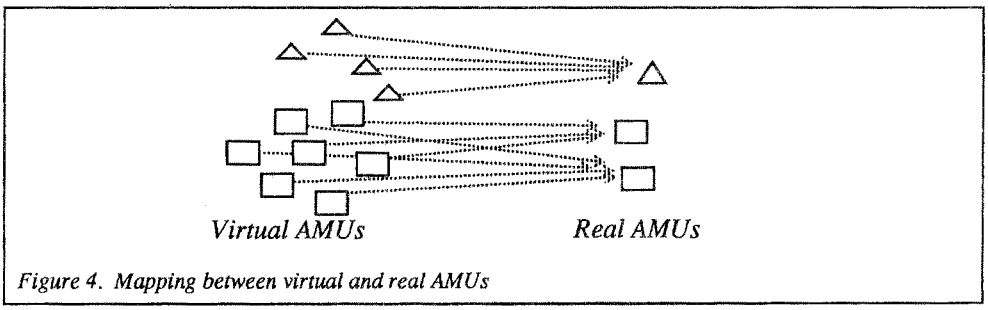

Figure 4. Mapping between virtual and real AMUs

The dependency graph of AMUs represents dependencies among virtual AMUs and will be used for creating a schedule.

A kind of *coloring algorithm* with some heuristics will be used for the scheduling. Each input of a virtual AMU could have two colors (*red, green* - for *wait, run*). The real physical AMU which is representing a virtual AMU, can start executing only if the virtual AMU has all inputs colored green.

The scheduling problems addressed by this paper

The scheduling problem of our synchronous multiprocessor system can be divided into two problems:

The first problem is *scheduling of instructions*, i.e. scheduling of the activity of all virtual AMUs. This scheduling consists in finding the right time for moving data from outputs of some AMUs to inputs of other AMUs. The schedule is a plan of actions executed by two CMUs. The result of this parallel activity must be the same as if the AMUs would execute serially. This semantics is defined by the dependency graph.

The second problem is the *allocation of the real AMUs*. This is solved after the scheduling of instructions.

Problems caused by control structure are discussed in [Kro89], and not dealt with in any detail here. The basic control structure is the conditional move, used in the same way as in [Tab87].

Data structures used by the scheduling algorithm

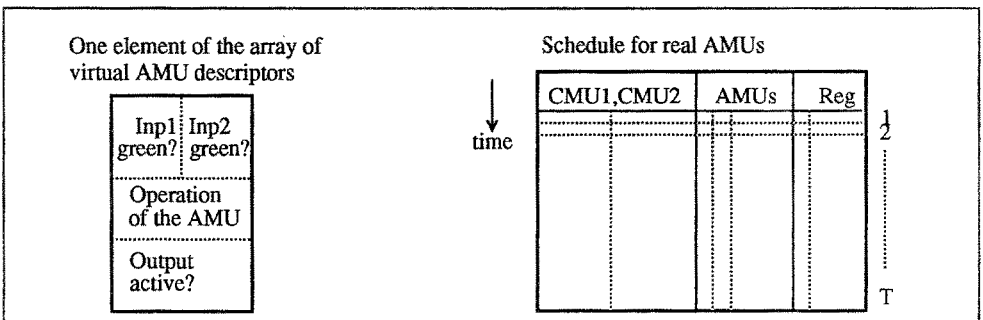

Figure 5. The data structures used by the scheduling algorithm

An important data structure is an array of AMU descriptors where the current state of each virtual AMU is stored. In the following description a matrix representation of the dependency graph of AMUs will be used. In this matrix G an element G[i,j] = n, if the virtual AMU[i] after its execution will color n inputs of AMU[j] green. Further an array of constants C will be used, which has as many rows as there are input constants for the given algorithm, and as many columns as there are virtual AMUs. An indication of computed results will be provided in an array R which has one element for each virtual AMU.

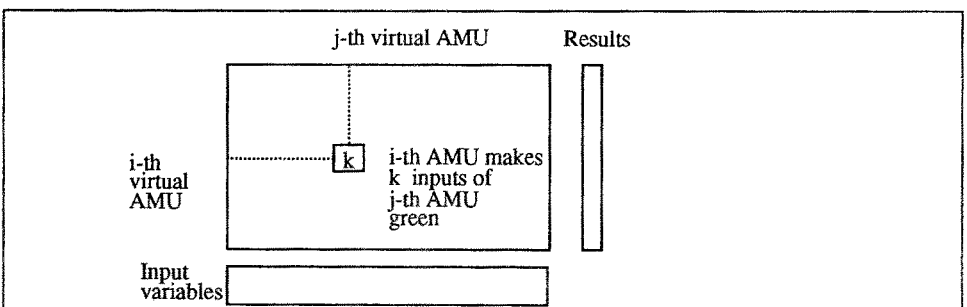

Figure 6. Representation of the dependency graph of units

A scheduling algorithm

In this section some properties of the algorithm and of the SIC machine determining the scheduling will be assumed. Remember that scheduling is for a synchronous SIC machine with multiple functional units. The absence of asynchronous waits has to be managed by smart scheduling and some possible insertion of delay instructions.

- The algorithm is restricted to IL program sections corresponding to basic blocks.
- We address *static scheduling* only, i.e. not dependent on the input data.
- We first schedule the *body of a loop* and then handle the whole loop as a unit for the next step of scheduling.

- We assume that the *execution time for each AMU* is known. For some specific cases (e.g. sin, cos) this time must be estimated.

We are scheduling AMUs only. Additional information from the intermediate language representation of the program will be used for code generation.

The algorithm:

Before we start the algorithm, *the longest path* in the IL program from the input values to the results must be found. There are several known methods to do this. This algorithm uses a few heuristics in order to improve the generated schedule.

The algorithm starts with all inputs of virtual AMUs colored red.

1 Apply constants to cover the inputs of AMUs to which they are connected. Then some inputs will become green, and some AMUs will be ready to start.

2 Build a list of AMUs that are ready to execute, i.e. a list of virtual AMUs which have all of their inputs green, and which have not already been started.

3 Order this list in decreasing order by the execution time needed for running each AMU beginning with the AMUs which are on the longest path. This step represents a *heuristic in the algorithm.* If a better heuristic for choosing an AMU from a list of free AMUs should be found, it should be used in this step. This *heuristic* assumes that it is better to start AMUs that take longer early, since more time is available then.

4 Pick the first AMU from the list.

5 Find an idle real AMU of the same type as the chosen virtual AMU in the schedule. If there is no one, then consider the next virtual AMU from the list of free AMUs. This is the second *heuristic*. Another heuristic could be to first search in the schedule and investigate if it would be better to wait for some AMU with a long execution time that should finish soon. If no one of the virtual AMUs in the list can be assigned to a real AMU, then wait for a message from the schedule that some AMU of an appropriate type finished. Then try again. This is the third *heuristic* feature. Another alternative is to backtrack as in the PROLOG implementations and to continue searching for a solution without waiting time.

6 When a real AMU has been chosen, then check if its output is free, i.e. if there isn't any result waiting to be used later. If the output is not free, search for the next real AMU, but with a free output. If there is no one, then generate a MOVE instruction from the output of the AMU to a register. The scheduler must remember the number of the virtual AMU to which the value stored temporarily into this register should be moved later. This is another *heuristic* (the fourth), because it would perhaps in many cases be better to wait in the hope that during the next time period storing to registers will not be necessary. The time spent on register management could be greater than the waiting time for a more suitable situation.

7 Schedule the feeding of input values to the chosen AMU, i.e. the action of one or two CMUs (due to the arity of the chosen AMU). Two CMUs can work in parallel because of the usage of a two-port memory in the extended SIC machine.

8 Schedule an activity of the started AMU.

9 Make one step in the time scale of the schedule.

10 Check the schedule if some AMU has finished execution at the current time.

11 Update the state of inputs and outputs of AMUs due to the dependency graph of AMUs.

12 Check if all results are green. If so, the final state has been reached, if not, then continue by step 2.

Comments:

This scheduling algorithm will first be used without any limitations in resources, i.e. potentially as many real AMUs as are needed for achieving the maximum degree of parallelism. The waiting time of both CMUs will be minimal. In this case no registers for storing intermediate results are needed. The customizable compiler informs the user about the first schedule found under the given conditions, i.e. unlimited resources and the simplest heuristics used. One part of the schedule is information about the number of cycles needed for reaching the final state. It represents the minimal execution time which can be achieved by using this architecture for a given algorithm with described scheduling heuristics.

The *quotient* between the number of virtual AMUs and the number of used real AMUs can be used as an easy computable *degree of parallelism of an algorithm* for the purposes of design of a SIC machine. If the dependency graph of variables corresponds to a *total order* relation then obviously one real AMU (of each type) will be enough and the degree of parallelism is one, i.e. such an algorithm is sequential. This quotient can be used in heuristics for an evaluation function.

The user can ask for all other schedules which can be derived under the same conditions by backtracking the algorithm (in the PROLOG implementation) for finding a minimal scheduling time. There can be a large number of these. Finding all such schedules can result in a combinatorical explosion. Because of heuristics it is not certain if a minimal schedule has been found until all schedules have been generated.

If the user is interested in the generation of a *specialized SIC machine* the compiler can systematically, step by step, decrease the number of used real AMUs and provide new schedules for the user to decide what combination of real AMUs would be most convenient. If the user could precisely formulate the *cost function* of this process (the dependence between the price of the realization and the speed of processing) then this can be an automatic process.

The user could also specify certain limitations concerning the set of real AMUs (e.g. to use a smaller number of AMUs of a specified type) and let the compiler compute the scheduling again from the beginning. After the user is satisfied with the schedule for a limited set of AMUs, he can specify limitations for the number of registers and let the compiler schedule again.

This whole process can be repeated for combinations of different heuristics. It is obvious that a heuristic which could give the best results for one case of resource limitation need not give the best results for another case of resource limitation. The places for including heuristics are marked in the algorithm and from the number of used heuristics at each point in the algorithm and from the number of cases of resource limitations we can derive an upper limit of the number of possible schedules. The danger of combinatorical explosion is of course very actual.

After the dialogue between the user and the compiler is completed, there is a fixed set of real AMUs, a set of registers and a schedule for the parallel activity of the AMUs.

The compiler generates code for the extended SIC machine by using the intermediate program representation and the created schedule table. Code will be generated for two CMUs working in parallel, and having access to a two-port memory.

If a method could be devised for splitting the set of variables into more than two memory banks, a set of SIC machines could be used and the performance of a vector machine could be achieved.

The usage of AMUs with pipelining

An interesting possibility occurs when the used real AMUs are designed for a pipeline mode of execution. Then all virtual AMUs of a certain type can be mapped into only one real AMU of the same type. Registers for saving intermediate results will be necessary in this case.

Conclusions

The problems of communication overhead and effective parallelism are serious problems, and they are likely to limit multiprocessors to relatively few processors in practical systems. Exploitation of multiprocessors depends strongly on finding ways to:

- Map serial programs to parallel programs.
- Identify useful parallelism, as opposed to parallelism that leads to wasted effort.
- Reduce overhead for scheduling tasks.

We have addressed the last problem by a method for compile time scheduling of multiple arithmetic processors in the single instruction machine architecture. It was demonstrated that the usage of two CMUs as the only means for communication and synchronization among processors can be efficient enough for a large class of tasks, in the case when there are operations with a longer execution time than the MOVE operation. This condition will be often valid when we consider some complex arithmetical operation with integers or floating-point operations. The described scheduling cross-compiler will not only generate code, but also will generate information about the degree of parallelism found in the translated algorithm, and of the most suitable structure for a specialized algorithm-tailored SIC machine and its performance.

In the given example we have compared the performance of the proposed machine scheduled by the described compiler with the performance of the I80286 executing the code produced by the Turbo-Pascal 5.0 compiler. We have assumed that the time needed for the execution of a MOVE operation on the SIC machine will be the same as for the I80286. Actually, it should be shorter because the time necessary in the I80286 for decoding the opcode will not be needed by the SIC and because of simpler addressing modes.

There are several possibilities for extensions and future work:

- Making a comparative study of the used heuristics in relation to other methods, and integrating with results from the field of parallelizing VLIW compilation.
- The possibility of building a *vector machine* from many SIC machines. Each memory bank of the vector machine can be represented by a data memory of one SIC machine which is the way how to get a SIMD machine based on the SIC architecture.
- The possibility of building a *systolic array machine* from many SIC machines. Each of these SIC machines must include AMUs with the operations "to *send message*" and "*to receive message*". However, there is a synchronization limitation. The SIC machine can not provide asynchronous waiting like a transputer.
- The possibility of building a hierarchical system of SIC machines. Each CALL instruction of the intermediate language representation of the program could be substituted by starting a SIC machine with this subroutine in its Instruction Memory. Then there could not only be more AMUs executing in parallel but also far more SIC machines working in parallel. The scheduling principles will be the same assuming that the execution time of the subroutine is known or some upper limit can be estimated.

There are no absolute rules which say that one architecture is better than another. The objective is to look both at cost and performance, not performance alone, in evaluating architectures.

Appendix A - A program example

We will present an example of the scheduling of the code of a body of a boolean function used in a program which determines if two line segments are intersecting each other. This function has not been specially written for the purpose of this illustration, it has been taken from the collection of algorithms in the book [Sed88],p.350-351. The complete environment of this function will be shown, but for our purposes it is enough to investigate the body of the function.

For this function body the translation into assembly language for the processor I80286 (IBM PC/AT) is appended. It was obtained from a Turbo Pascal 5.0 compilation. The given number of machine cycles for each instruction has been calculated accordingly to [MorAlb86]. The listing of assembly instructions has been obtained using the Turbo Debugger.

```
program inter;
 type
     point = record
                 x,y : integer;
             end;
     line  = record
                 p1,p2 : point;
             end;
 var
     ln1,ln2 : line;

function CCW(p0,p1,p2 : point) : integer;
 var
     dx1,dx2,dy1,dy2 : integer;
     c,d : integer;
 begin

{  Pascal              code for I80286        number of cycles  }
{-------------------------------------------------------------}
   dx1:=p1.x-p0.x;
                    mov     ax,[bp+08].................3
                    sub     ax,[bp+0C].................7
                    mov     [bp-04],ax.................5
   dx2:=p2.x-p0.x
                    mov     ax,[bp+04].................3
                    sub     ax,[bp+0C].................7
                    mov     [bp-06],ax.................5
   dy1:=p1.y-p0.y;
                    mov     ax,[bp+0A].................3
                    sub     ax,[bp+0E].................7
                    mov     [bp-08],ax.................5
   dy2:=p2.y-p0.y;
                    mov     ax,[bp+06].................3
                    sub     ax,[bp+0E].................7
                    mov     [bp-0A],ax.................5
   c:=dx1*dy2;
                    mov     ax,[bp-04] ................ 3
                    imul    word ptr [bp-0A] ........ 24
                    mov     [bp-0C],ax ................ 5
   d:=dy1*dx2;
                    mov     ax,[bp-08] ................ 3
                    imul    word ptr [bp-06]......... 24
                    mov     [bp-0E],ax ................ 5
   if c > d then CCW:=1;
                    mov     ax,[bp-0C] .................3
                    cmp     ax,[bp-0E] .................7
                    jle     Inter.25... ..............12
                    mov     word ptr [bp-10] ......... 5
   if c < d then CCW:=-1;
                    mov     ax,[bp-0C] ................ 3
                    cmp     ax,[bp-0E] ................ 7
                    jnl     Inter.26  .. ..............12
                    mov     word ptr [bp-12].......... 5
   if c = d then
                    mov     ax,[bp-0C] ................ 3
                    cmp     ax,[bp-0E] ................ 7
                    jne     Inter.31  .................10
     begin
       if (dx1*dx2 < 0) or
          (dy1*dy2 < 0) then
                CCW:=-1 else
                    mov     ax,[bp-04] ................ 3
                    imul    word ptr [bp-06]..........24
                    or      ax,ax ..................... 2
                    jl      007A .....................10
```

```
                mov     ax,[bp-08] ..............  3
                imul    word ptr [bp-0A].........24
                or      ax,ax ...................  2
                jnl     Inter.29  ...............10
                mov     word ptr [bp-12].........  5
                jmp     Inter.31  ...............10

    if (dx1*dx1 + dy1*dy1) >=
       (dx2*dx2 + dy2*dy2) then
            CCW:=0 else CCW:=1;
                mov     ax,[bp-0A] ..............3
                imul    word ptr [bp-04].........24
                mov     cx,ax....................  2
                mov     ax,[bp-06] ..............  3
                imul    word ptr [bp-08].........24
                add     ax,cx ...................  2
                mov     bx,ax ...................  2
                mov     ax,[bp-08] ..............  3
                imul    word ptr [bp-06].........24
                mov     cx,ax ...................  2
                mov     ax,[bp-04] ..............  3
                imul    word ptr [bp-0A].........24
                add     ax,cx ...................  2
                cmp     ax,bx ...................  2
                jl      00AE ....................10
                xor     ax,ax ...................  2
                mov     [bp-02],ax ..............  5
                jmp     Inter.31 ... ............12
                mov     word ptr [bp-10] ........  5
    end;
end
```

To process the body of function CCW (because the relation between c and d is data dependent, we consider all cases) about *440 cycles are necessary for the code of the processor 80286* which is used as a comparison.

Appendix B - The example translated into the IL

For the purposes of this presentation a very simple stack-oriented intermediate language has been chosen. The translated example is presented below.

pusha ...push an address on the compilation stack.
pushc .. push a constant on the compilation stack
add, sub, mul, st, or, comp ....generate code for an operation over the elements described in the stack
ifj....generate an instruction for "jump if false", i.e. if the previous comparison sets the flag to false

```
usha      dx1
pusha     p1.x
pusha     p0.x
sub
st
----------

similar to the previous
.
----------
pusha     c
pusha     dx1
pusha     dy2
mul
st
----------
.
similar to the previous
.
----------
pusha     c
pusha     d
comp      >
ifj       L1
----------
pusha     ccw
```

```
pushc     1
st
----------
lab       L1
pusha     c
pusha     d
comp      <
ifj       L2
----------
pusha     ccw
pushc     -1
st
----------
lab       L2
pusha     c
pusha     d
comp      =
ifj       L3
----------
pusha     dx1
pusha     dx2
mul
----------
pushc     0
comp      <
pusha     dy1
pusha     dy2
```

```
mul
----------
pushc     0
comp      <
or
----------
pushc     1
comp      =
ifj       L4
----------
pusha     ccw
pushc     -1
st
----------
lab       L4
pusha     dx1
pusha     dx1
mul
----------
pusha     dy1
pusha     dy1
mul
add
----------
pusha     dx2
pusha     dx2
```

```
mul
--------------
pusha     dy2
pusha     dy2
mul
add
--------------
comp      >=
ifj       L5
--------------
pusha     ccw
pushc     1
st
--------------
la        L5
pusha     ccw
pushc     0
st
--------------
lab       L3
```

Appendix C - A dependency graph based on the program in IL form.

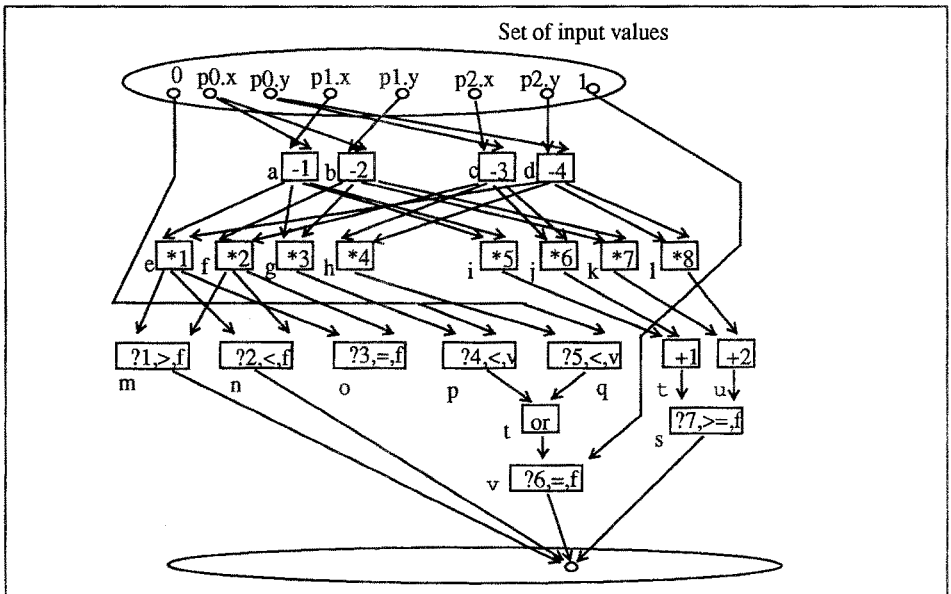

Figure 7. A dependency graph of arithmetic processor units (AMUs)

A matrix representation of the dependency graph.

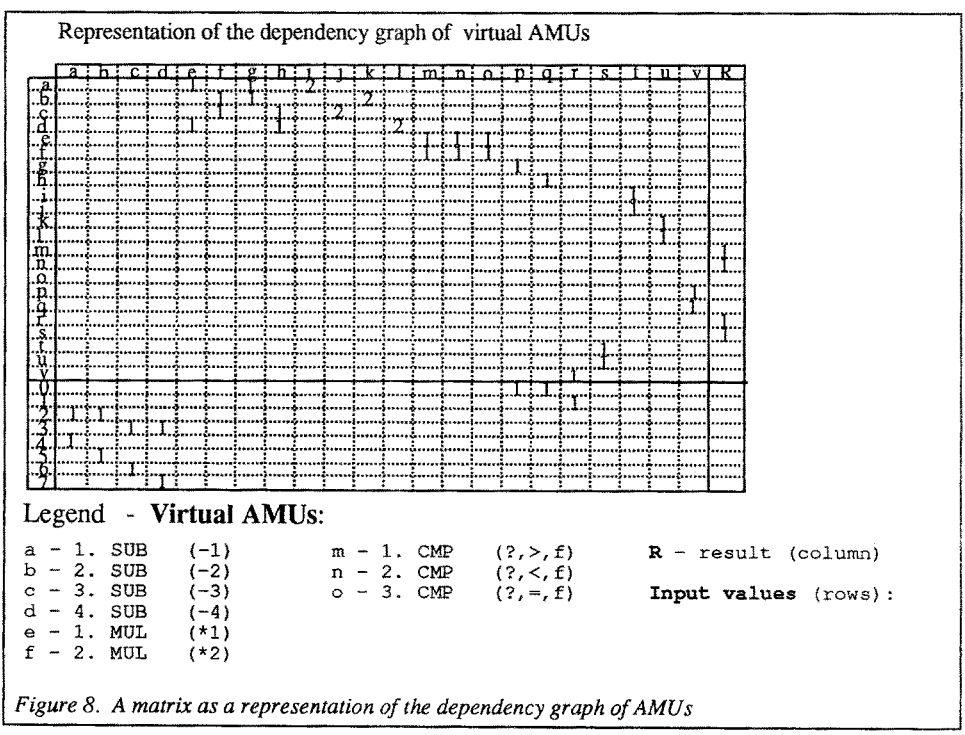

Figure 8. A matrix as a representation of the dependency graph of AMUs

Appendix D - Derived schedules for the program example

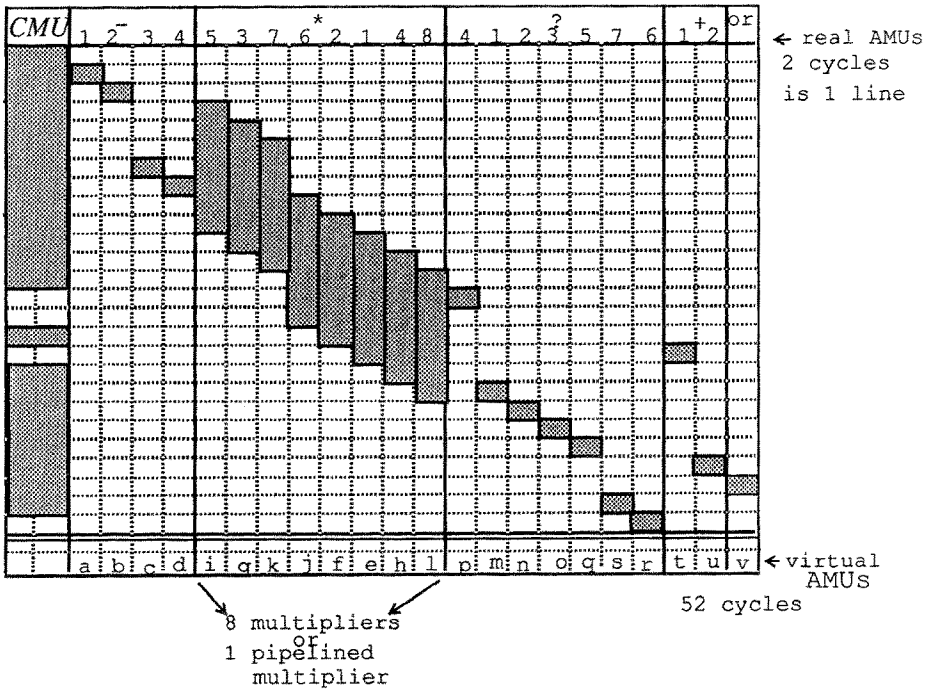

Figure 9. A derived schedule for real AMUs without limitation of resources

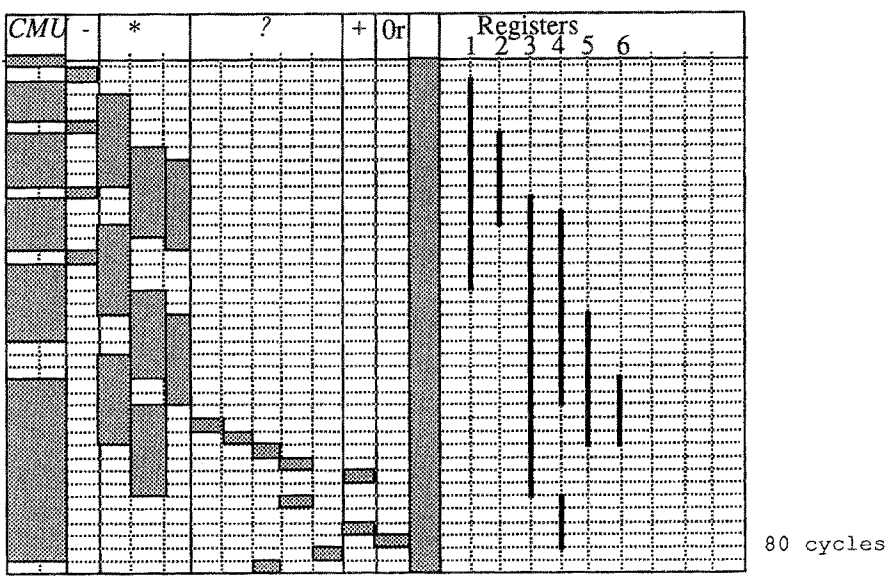

Figure 10. A schedule for limited resources: 1 subtract, 3 multiply, 5 conditional, 1 addition, and 1 logical-or AMU.

Appendix E - Generated SIC machine code for the program example

```
       CMU - 1:                          CMU - 2:

  p1.x  -->  inp1(-1)                 p0.x  -->  inp2(-1)
  p2.x  -->  inp1(-2)                 p0.x  -->  inp2(-2)
out(-1) -->  inp1(*5)           out(-1)  -->  inp2(*5)
out(-1) -->  inp1(*3)           out(-2)  -->  inp2(*3)
out(-2) -->  inp1(*7)           out(-2)  -->  inp2(*7)
  p1.y  -->  inp2(-3)               p0.y  -->  inp2(-3)
  p2.y  -->  inp1(-4)               p0.y  -->  inp2(-4)
out(-3) -->  inp1(*6)           out(-3)  -->  inp2(*6)
out(-3) -->  inp1(*2)           out(-2)  -->  inp2(*2)
out(-3) -->  inp1(*1)           out(-4)  -->  inp2(*1)
out(-1) -->  inp1(*4)           out(-4)  -->  inp2(*4)
out(-4) -->  inp1(*8)           out(-4)  -->  inp2(*8)
out(*3) -->  inp1(?4)                #0  -->  inp2(?4)
    Rw1 -->  Rw1                    Rw2  -->  Rw2
    Rw1 -->  Rw1                    Rw2  -->  Rw2
out(*5) -->  inp1(+1)           out(*6)  -->  inp2(+1)
    Rw1 -->  Rw1                    Rw2  -->  Rw2
out(*1) -->  inp1(?1)           out(*2)  -->  inp2(?1)
out(*1) -->  inp1(?2)           out(*2)  -->  inp2(?2)
out(*1) -->  inp1(?3)           out(*2)  -->  inp2(?3)
out(*4) -->  inp1(?5)                #0  -->  inp2(?5)
out(*7) -->  inp1(+2)           out(*7)  -->  inp2(+2)
out(?4) -->  inp1(or)           out(?5)  -->  inp2(or)
out(+1) -->  inp1(?7)           out(+2)  -->  inp2(?7)
out(or) -->  inp1(?6)                #1  -->  inp2(?6)
-------------------------------------------------------
out(?1) -->  Rw1                out(?2)  -->  Rw2
    #L1 -c>  PC1                     #L2 -c>  PC2
     #1 -->  CCW                     #-1 -->  CCW
  #END1 -->  PC1                  #END2 -->  PC2
L1:                             L2:
out(?6) -->  Rw1                out(?7)  -->  Rw2
    #L3 -C>  PC1                     #L4  -c>  PC2
    #-1 -->  CCW                      #1  -->  CCW
  #END1 -->  PC1                  #END2  -->  PC2
L3:                             L4:
END1:                                #0  -->  CCW
   #END -->  PC2                END2:
   #END -->  PC1                   #END  -->  PC1
                                   #END  -->  PC2          61 cycles
```

Figure 11. A scheduled program for the designed parallel SIC machine given unlimited AMU resources.

REFERENCES

[AzTab80] Azaria ,H.,Tabak,D.: Bit-Sliced Realization of a CMOVE Architecture Microcomputer. In: EUROMICRO Journal, Vol.6, No.6, pp.373-380, Nov.1980.

[AzTab83] Azaria,H.,Tabak,D.: Design Consideration of a Single-Instruction Microcomputer - A Case Study. Microprocessing and Microprogramming, Vol.11, No.3,4, pp.187-194, North-Holland, 1983.

[BlaKro87] Blazek,Z.,Kroha,P.: Design of a Reconfigurable Parallel RISC Machine. EUROMICRO'87, Portsmouth, in: Microprocessing and Microprogramming, Vol.21., No.1-5, pp.39-46, North-Holland, 1987.

[Fi84] Fisher,J.A.,Ellis,J.R.,Ruttenberg,J.C.,Nicolau,A.: Parallel Processing: A Smart Compiler and a Dumb Machine. In:Proceeding of the Compiler Construction, pp.37-47, Association for Computing Machinery, June 1984.

[Fri83] Fritzson,P.: Symbolic Debugging Through Incremental Compilation in an Integrated Environment. The Journal of Systems and Software 3, pp.285-294 (1983).

[Fri86] Fritzson,P.: A Common Intermediate Representation for C, Pascal, Modula-2 and FORTRAN-77. LITH-IDA-R-86-38, Research Report, PELAB, December 1986, University of Linköping.

[Kess82] Kessler,P.B.: The Portable C Compiler's Intermediate Representation, as Used by the Berkeley Pascal Front-End for the VAX. (An unpublished paper).

[Kro86] Kroha,P.: Design of a Code Generator by Help of a PROLOG-Database. Proceedings of the Workshop Compiler Compilers and Incremental Compilation, Bautzen , October 1986, in: IIR, 12/86, AdW.

[Kro88] Kroha,P.: Code Generation for a RISC Machine. Proceedings of CCHSC'88, Berlin , GDR, October 1988. In Lecture Notes on Computer Science 371, Springer Verlag.

[Kro89] Kroha,P.: An Extension of the Single Instruction Machine Idea. LITH-IDA-R-89-25, Research Report, PELAB, June 1989, Department of Computer Science, Linköping University, Sweden.

[Kuck72] Kuck,D.J.,Muraoka,Y.,Chen,S.C.: On the Number of Operation Simultaneously Executable in Fortran-like Programs and their Resulting Speedup. IEEE Transactions on Computers C-21(12), pp.1293-1310, December, 1972.

[Kuck78] Kuck,D.J.: The Structure of Computers and Computations. John Wiley and Sons, New York, 1978.

[Lip76] Lipovski,G.J.: The Architecture of a Simple, Effective, Control Processor. EUROMICRO'76, in: Microprocessing and Microprogramming, pp.7-18, North-Holland, 1976.

[Mil86] Milutinovic,V.M.: RISC Architecture. Tutorials, EUROMICRO'86, North-Holland.

[MorAlb86] Morse,S.P.,Albert,D.J.: The 80 286 Architecture. John Wiley & Sons,1986.

[Nam88] Namjoo,M.,Agrawal,A.: Implementing SPARC: A High-Performance 32-Bit RISC Processor. Sun Technology, Winter 1988.

[Pat82] Patterson,D.A.,Sequin,C.H.: A VLSI RISC. IEEE Computer, September 1982.

[Po88] Polychronopoulos,C.D.: Parallel Programming and Compilers. Kluwer Academic Publishers, 1988.

[Sed88] Sedwick,R.: Algorithms. Addison-Wesley, Second edition, pp.350-351,1988.

[TabLip80] Tabak,D.,Lipovski,G.J.: MOVE Architecture in Digital Controllers. IEEE Trans. Comput., Vol. C-29, pp. 180-190, Feb.1980.

[Tab87] Tabak,D.: RISC Architecture. Research Studies Press, John Wiley & Sons, 1987.