The SPARK 2.0 system*— a special purpose vector processor with a VectorPASCAL compiler

A. FORMELLA†, A. OBÉ†, W.J. PAUL, T. RAUBER†, D. SCHMIDT†

Computer Science Department

Universität des Saarlandes

D–6600 Saarbrücken, Germany

+49–681–302–2436

wjp@cs.uni–sb.de

Abstract

This paper describes the architecture of the SPARK 2.0 processor, introduces a compiler for VectorPASCAL and outlines a few interesting details. Features of the architecture are the flexible address generation during vector operations and the large memories closely connected to the functional units. The source language allows to write programs with vector statements avoiding scalar inner loops. The compiler employs several optimizing strategies to utilize the architectural benefits efficiently.

1 Introduction

In the scientific world there exist many applications (e. g. molecular dynamics) with extensive use of index table driven algorithms. To achieve high sustained/peak performance ratio on such non–standard addressed vector operations we have built a special node processor. An operating system and a compiler for VectorPASCAL has been implemented supporting

The SPARK 2.0 processor is a further development of the SPARK 1.0 processor built by a project group of IBM Almaden Research Center [APB87]. It consists of three functional units

*joint study agreement with IBM Almaden Research Center

†research partially funded by DFG, SFB 124

(branch, fixed point and floating point) extended by a main memory and an interface board. Concurrent operation of these units allows for overlapping address calculation and floating point operations (64–bit IEEE) in the manner of vector processing.

The prototype of SPARK 2.0 is connected to a HOST. The operating system is shared between them performing monitoring, file transfer, graphic extension and application control. We have implemented a compiler for the SPARK 2.0 processor that uses VectorPASCAL as source language. The compiler generates SPARK 2.0 assembly code which is postprocessed by an assembler. Figure 1 gives an overview of the SPARK 2.0–system.

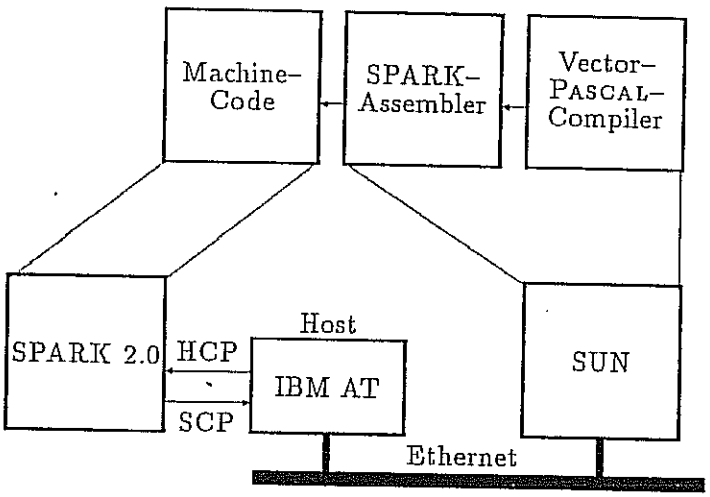

Figure 1: SPARK 2.0–system

Besides the usual vector extensions (see FORTRAN 8X, VECTRAN [Pa82], ACTUS [PCM83]), VectorPASCAL especially supports the building and usage of index vectors. The compiler is tuned to exploid the special features of the SPARK 2.0. The core of the compiler is an optimizer that tries to utilize the floating point memory efficiently. This optimizer includes an algorithm to generate optimal evaluations of vector expression trees (the algorithm for scalar trees does not the job) and a heuristic to find the best ratio of spilling to strip mining. By the time we are developing a trace scheduling extension of the compiler to execute as much operations in parallel as possible.

The sustained/peak performance ratio of the SPARK 2.0 is very high for e.g. molecular dynamics and Livermore Loops, kernels 13 and 14. On these applications a CRAY–1, e.g., only reaches a ratio of about 10 %. The peak performance of the SPARK 2.0 is about 10 MFlop/s.

Section 2 gives a short description of the architecture of the processor and its environment. In section 3 we introduce the system software we have implemented to work with the prototype. It's

a shared operating system developed in C and in SPARK 2.0–assembly language. The compiler is described in section 4. The key point of the description is how to manage index vectors. Some sophisticated details of the implementation such as run–time stack management and code loading are explained in section 5.

2 Architecture of the SPARK

The SPARK 2.0 node processor is built out of five functional units:

- branch unit (BU)
- fixed point unit (XPU)
- floating point unit (FPU)
- main memory unit (MU)
- interface unit (IU)

The simultaneous operation of FPU, XPU and BU permits the overlapping of a floating point operation, non–trivial address generation for floating point operands (in the XPU), and instruction fetch. Those operations support scatter–gather algorithms but require only one instruction cycle ($100ns$). Scatter–gather operations between main memory and floating point memory involve the MU, the FPU, the XPU (generating the addresses for main memory), and the BU in parallel. For communication with the 'outside world' (IBM AT 03) the IU is implemented. A simplified block diagram of SPARK 2.0 is given in figure 2.

Branch Unit: The branch unit (BU) is responsible for branch, loop and interrupt handling. Instructions are fetched from a 18K instruction buffer (16K RAM and 2K ROM). The ROM contains the operating system and standard functions. One fourth of the RAM is meant to hold resident code, whereas the other three parts are managed by the application. Initially the RAM part has to be loaded with instruction code from main memory or interface memory using move–instructions.

Parts of the 64–bit instruction word are distributed to all other units. Two different instruction types should be distinguished (see figure 3): *compute* instructions and *move* instructions. *Compute* instructions can involve the branch unit, the fixed point unit, and the floating point unit simultaneously. *Move* instructions specify transfer operations between memories, and optionally a fixed point operation for index addressing.

Instruction fetch of the next operation is done during the execution of the previous one.

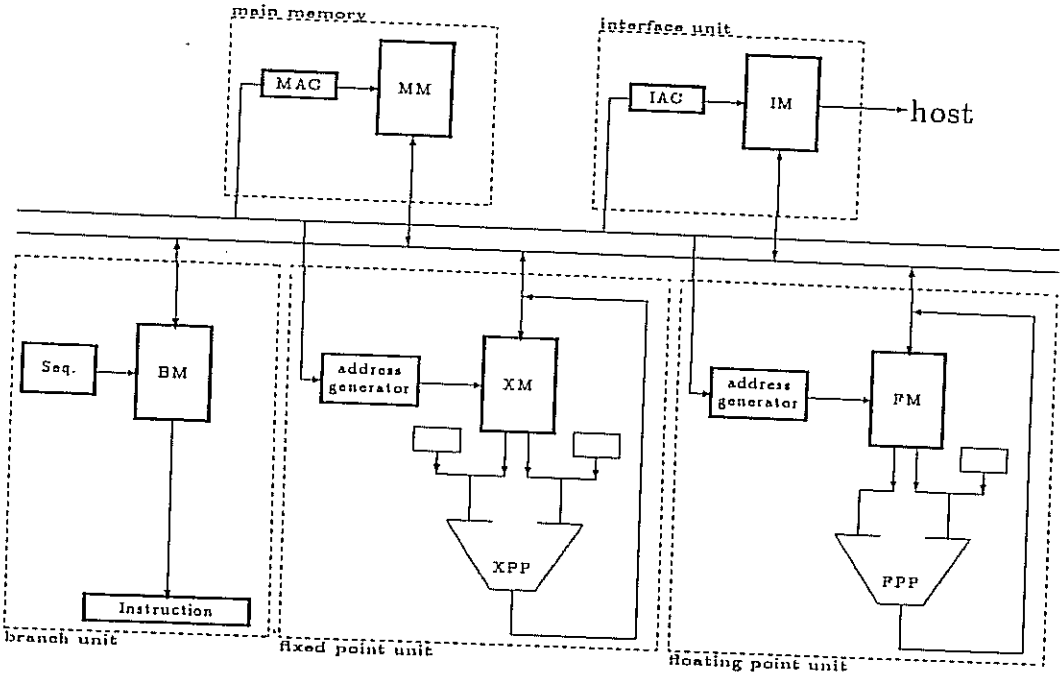

Figure 2: sketch of the architecture

However, there is no delay slot before a conditional branch. The condition code generated by a previously executed fixed point operation can direct a jump in the following instruction. Nevertheless, floating point conditions have to fall through the pipeline stages. Vector operations are performed by instruction loops. A loop count register determines how often the same instruction block is to be executed. If the block contains only one instruction it's called a single instruction loop. The least significant bits of the branch address can be specified by other sources than the program counter. For every machine cycle an instruction is fetched, for instance, coincidently generated condition code can select the next instruction to be fetched without changing the PC. This mechanism used in a single instruction loop supports vector instructions with different operations performed on the components (butterfly jump).

The internal stack of the branch unit with at most 33 entries can be used to nest loops or perform calls to subroutines. This allows to implement the inner loops of an algorithm with small overhead. For indirect jumps and variable loop length a branch register holds the value stored by a fixed point operation. Interrupts are possible on every instruction boundary. They are implemented as unexpected subroutine calls using the internal stack.

Fixed Point Unit: The fixed point unit is mainly devoted to generate non–trivial addresses for the floating point memory. A 4K word register file XM can be accessed as a vector register

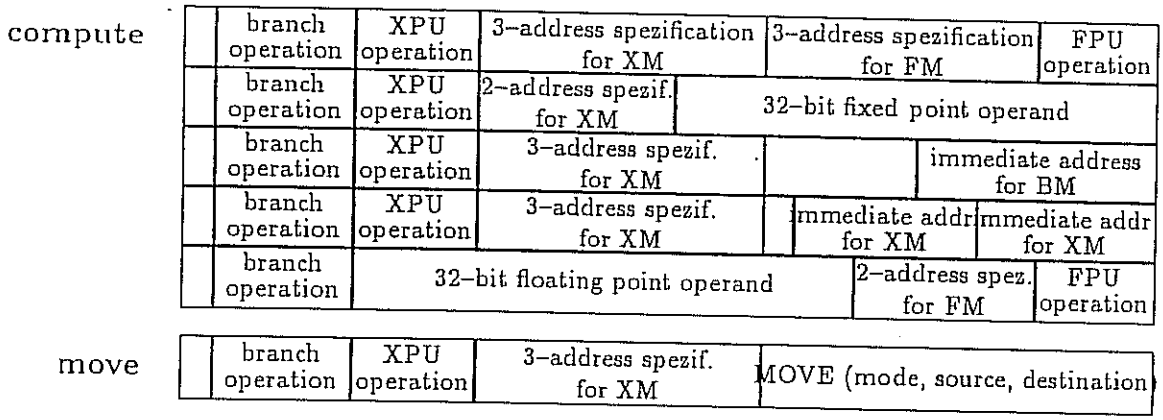

Figure 3: instruction formats

in order to support index–vector addressing for floating point operands and scatter–gather operations between main memory and floating point memory. Handling of large tables of indices, e. g. neighbor tables in molecular dynamics, may be very convenient. The fixed point processor consists of an ALU and a fixed point multiplier. The ALU works in a non–pipelined mode at a rate of 10 MOP/s. Only fixed point multiplications require an execution time of 200 ns (5 MOP/s).

Every unit is equipped with special purpose registers that may be computed by the fixed point unit. These registers can be used as left operand of the fixed point processor. In every instruction specifying no floating point operation the right operand may be an immediate 32–bit number. Figure 4 illustrates the main data paths of the fixed point unit.

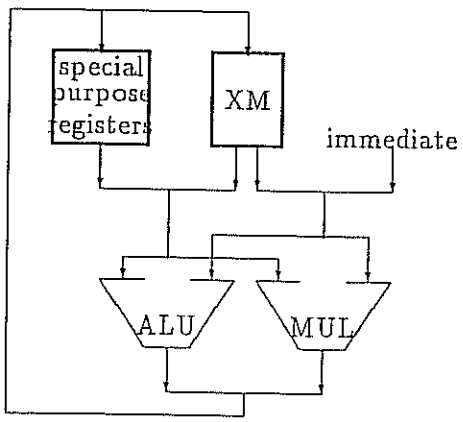

Figure 4: fixed point unit

A flexible addressing scheme for the three–ported register file permits fast access to consecutive vector elements and scalars. Two address registers XAC and XAC2 with optional autoincrement provide two different vector operands in one instruction (fixed point operations with three different vector operands or more complicated addressing may be computed by the floating point unit). Two registers XBASE0 and XBASE1 hold the base addresses for blocks of 16 scalars (each of the 16 scalars in a block can be immediately accessed without loss of a coincident floating point operation). Renouncing a parallel floating point operation, it is possible to address two of the 4096 registers immediately.

Floating Point Unit: The floating point unit conforms to the IEEE standard (ANSI/IEEE Std 745–1985). It is optimized for 64–bit floating point computation. The floating point processor consists of the WEITEK chip set (ALU and Multiplier/Divider). Both single– and double–precision ALU– and multiply–operations are executed in pipelined mode (six stages) at a rate of 10 MFLOP/s. Divide–operations are not pipelined and can only be executed at a rate of approximately 0.8 MFLOP/s for double–precision and 1.1 MFLOP/s for single–precision operands. Chained operations are not implemented because they are of little use in molecular dynamic simulations (we designed the processor especially for this application).

A three–ported 4Kx64 register file holds operands and results. Powerful addressing modes support both fast vector and scalar access. Three address registers with optional autoincrement or parallel load with a fixed point result support three–address vector operations. A special instruction field allows the 4K registers to be viewed as 16 vector registers of length $256, \ldots, 1$ vector register of length 4096, or as sets of 16 scalar registers. Thus, it it possible to handle up to three different vectors with the same stride $c > 1$ (or the same index vector) in one instruction. Two base–address registers FBASE0 and FBASE1 provide simultaneous access to two sets of scalars. Up to three different scalars may be specified in one instruction. One operand may optionally be specified as 32–bit number in the instruction word. This constant is automatically extended to a double precision number.

Main Memory Unit: As main memory (MM) 256K 64–bit words are available. MM is accessed using move–instructions. A move transfers a block of data or code to any other memory of the processor. Each machine cycle one item is moved. The start–up time of a move takes four cycles: two to specify the starting addresses of the blocks, one to load the length of the block and one for internal delays. Between MM and FM imove–instructions provide scatter–gather

operations. Such moves transfer one item every other cycle. If FM is involved the pipeline is drained first leading to a slidely greater start–up time.

Interface Unit: Four 16–bit bidirectional channels connect the SPARK 2.0 to the HOST. Two of them are used to exchange information via registers. In each direction there is a register which can be written by one side and be read by the other side. To avoid read–before–write the third channel provides some flags generated by hardware, but both read checked by software protocol and forced read can be used. The fourth channel is used by the HOST to access the dual–port memory the interface unit is equipped with. The buffer can hold 16K 64–bit words, so large blocks of data can be moved between SPARK 2.0 and HOST. On SPARK 2.0s side the memory is connected to the internal memory bus, and access is performed using move–instructions. While the buffer is retained by one side it can't be accessed by the other one, but overlapping of compute and communicate between HOST and SPARK 2.0 is still possible.

2.1 Index Addressing of Main Memory

The main memory address register MAC may be used either as source and destination register of the fixed point unit (XPU). It can be updated every machine cycle, so we have a very fast address generation (see figure 5 (left part) for the data paths). In the current implementation the random access time to main memory, however, is the bottleneck for indexed DMA tranfers. Thus, scatter–gather vector moves between main memory and floating point memory require twice as much time as fastest floating point vector operations.

2.2 Index Addressing of Floating Point Memory

The floating point memory requires three addresses for binary operations (a op b = c). Running at full speed, the floating point unit requires generation of these addresses in one machine cycle. The fixed point unit supplies only one completely general address calculation each machine cycle (see figure 5 (right part) for the data paths). This value may be propagated to any of the three address ports of the floating point memory. Analysis of scientific code, especially molecular dynamic code, indicates that in most cases at most one nontrivial address calculation is required and one fixed point unit is sufficient.

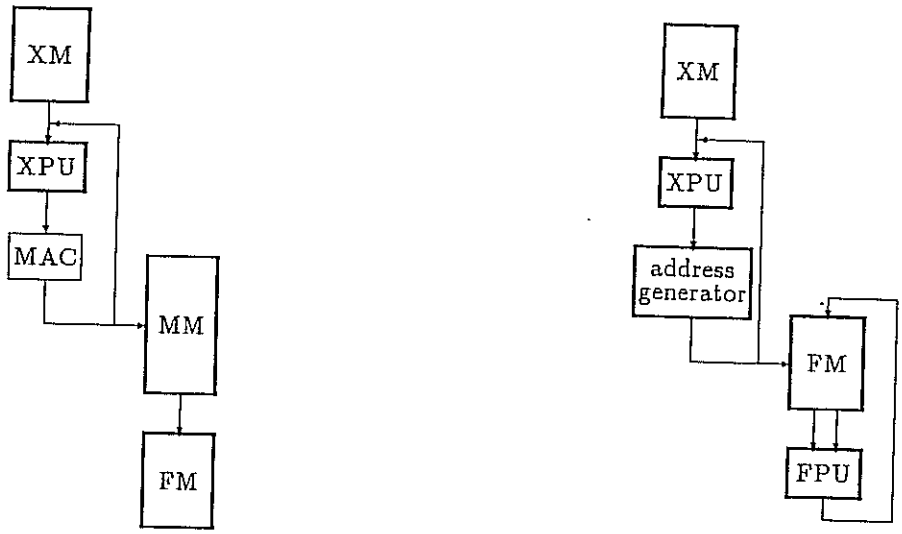

Figure 5: scatter–gather between main memory and floating point memory (left part) and scatter–gather between floating memory and functional unit (right part)

2.3 Fast Update of Tables of Indices

In the introduction we have mentioned that the SPARK 2.0 supports a fast update of neighbor tables. One application that might require such neighbor tables and their fast computation, is a class of fast algorithms for molecular dynamics simulation. Let's have a look at a set of identical particles (with N elements) and its interactions. The efficient summation over interactions is the key to several efficient molecular dynamics algorithms. By assuming that one has short–range interactions, the number of elements for summation may be reduced. Thus, for a given cutoff radius one can specify a neighbor table for each particle (the separation between a particle and an element of its neighbor table is less than the cutoff radius). Infrequent and fast update of neighbor tables is necessary for fast computation. Figure 6 illustrates how it may be computed by the SPARK 2.0 processor. For a given particle i the separations to all other particles j are stored into FM. Then, the FPU compares the separations r_{ij} $(1 < j < N)$ with the cutoff radius R in a *single instruction loop* (the butterfly mechanism selects between two instruction, but the PC is not changed). In parallel (in the same instruction loop!), the condition $r_{ij} < R$ is used to update the neighbor table of particle i in the fixed point unit. The address register (XAC) for the neighbor table increments only if a new neighbor was found ($r_{ij} < R$) and its index was written into XM.

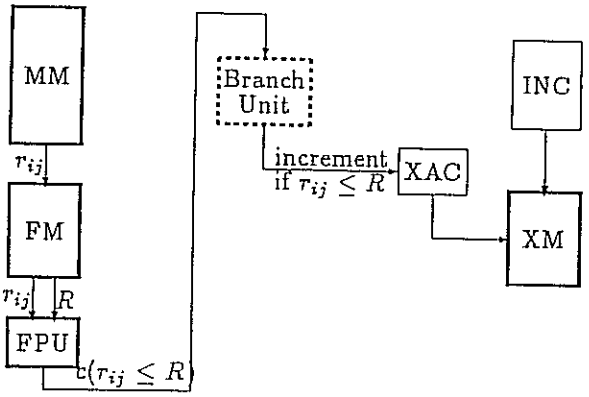

Figure 6: data paths during updating a neighbor table by a single instruction loop

2.4 Performance Considerations

In the year 1987 the processor SPARK 2.0 was designed with two restrictions: (1) worst case timing and (2) without custom chips. Therefore, only 10 Mflops were available by using a commercial floating point arithmetic unit (WEITEK WTL2264/2265–100). Nevertheless a high sustained/peak performance ratio for index table driven algorithms is reached so that we have a very powerful node processor for such applications. The SPARK 2.0 is mainly designed for molecular dynamic simulations. Thus, some features such as chaining are not implemented (although it would be possible) because of little use in molecular dynamics codes.

Now, we compare the power of the SPARK 2.0 architecture with that one of the CRAY–1 and of one node of an Intel Hypercube (equipped with an i860) by giving the sustained/peak performance ratio for the Lawrence Livermoore Loops (a common benchmark for supercomputers performance considerations). We have outlined the kernels with index addressing. Considering these kernels one can see the speed upgrade caused by a special architecture and not by a fast technology. The improvement shown in the last column of the table could be achieved by a further optimization: direct moves between FM and XM are not used by the compiler, but they could be implemented.

Kernel	CRAY–1		Intel		SPARK 2.0		SPARK 2.0[1]	
peak	80.0	100.0 %	26.00	100.0 %	9.60	100.0 %	9.60	100.0 %
13	3.3	4.1 %	1.77	6.6 %	1.62	16.9 %	1.88	19.9 %
14	7.7	9.6 %	2.16	8.1 %	2.82	29.4 %	3.25	33.8 %

On a very popular problem, namely computing MANDELBROT's set, the processor reaches a

sustained/peak performance ratio of 100 %, wheras no start–up time is considered (e.g. assume 1000 iterations on every point of the calculated area). The reason for that high ratio lies in the indexing capability together with the butterfly jump. Consider the following VectorPASCAL–sequenz (for an explanation of the syntax see 4):

```
for i := 1 to indexlength(IM) do
  if A2_B2[i] < 4.0
     then IN := IN || IM[i];
IM := IN;
```

Before entering the loop the index vector IM holds all indices of the points which have been iterated last. The if–statement determines wether they still have to be iterated or wether they have to be dropped. After the loop the index vector IN holds all indices of the points which participate on further iterations and he is assigned to IM. The loop can be implemented in SPARK 2.0–assembly language as a single instruction loop performing a butterfly jump. Thus the body of the loop executes one maschine instruction for each i.

3 System–Software on the prototype

The operating system is shared between SPARK 2.0 and HOST. On SPARK 2.0's side the Spark Control Program (SCP) provides basic functions to communicate with the HOST, to transfer data and to start and monitor application programs. On the HOST the Host Control Program (HCP) deals with the file system, user actions and input/output functions ([Fo90]). As extended software we have implemented a set of trigonometric functions and some basic graphic routines. SCP and HCP are based on versions written by Chris Lutz at IBM Almaden Research Center as control software for the SPARK 2.0 1.0 prototype ([Lu87]).

3.1 Spark Control Program

SCP is dedicated to provide simple functions controlling the vector processor. An important issue has been the implementation of routines which allow for debugging and testing during the built–up phase of the prototype. SCP handles the interface channels, interrupts caused by errors or user actions and system calls performed by an application. Main part of SCP is the command interpreter which allows to execute a lot of commands such as echoing on interface channels,

moving blocks of data to the HOST and visa versa, starting programs, whose initial sequences have been loaded previously to BM and some special ones dealing with specific registers.

After reset the bootstrap routine sets the processor to a defined state and waits for any command sent by the HOST. Such a command with its parameters is passed through one of the interface channels and the hand–shake is accompanied by various protocols. All data transfers between HOST and SPARK 2.0 are varified on the HOST's side by reading back the written blocks. Beside executing commands sent by the HOST, SCP can deal with commands required by an application program encoded in a function number and a parameter block. In most cases the command is passed to the HOST. In the case of a graphic function it is directly executed (see below).

The code is located within the ROM–part of BM (together with the trigonometric functions it occupies less than 1500 words). It uses the topmost 64 registers of XM as system variable block. This block is used to determine what is going on during the execution of any application program. Any entrance to SCP — whether as application call or as interrupt — saves all special purpose registers into this area, so HCP can inspect the state of the processor by simply reading this part of XM. The command interpreter of SCP can be extended into the RAM–part of BM, so further features are easy to add. The interrupt entry table, which contains the starting points of the service routine, can be located in the RAM too, so a new, more flexible interrupt handling can be provided if necessary.

A special feature of SCP is its capability to single step through an application program. After the execution of every instruction an interrupt service routine is entered without changing the context of the application. Now, the control is given to HCP, so the user can examine and manipulate by several commands the processors state. The RFI–command returns to the application previously restoring the saved information. The pipeline of the floating point unit is frozen during that single step interrupt, which allows for stepping through vector operations too. However, the unit can be unfrozen and it is possible to take a close look at the content of the pipeline.

3.2 Host Control Program

HCP is implemented in C as a counterpart to SCP on the HOST. The lowest level performs the hand–shake protocols on the interface channels. On the highest level commands arriving from three sources are executed. The first source is the user typing in his commands on the

keyboard. Sequences of commands can be put into command files which are executed typing in the dofile–command. Such a batch job — which can be seen as the second command source — is always interruptable and saves a lot of "keypressing" time. As third source SPARK 2.0 itself is able to pass system calls of an application mostly requiring access to a file or printing messages on the screen. All three sources are examined in turn, so every request can be satisfied as early as possible. A special escape–command generates a new level within the execution hierarchy; all pending commands were pushed aside and HCP waits on actions of the user. For instance he wants to inspect some memory locations or register contents. Resuming and continuing the old command sequence is as possible as simply aborting it.

On the one Hand HCP provides standard input and standard output functions used by application programs on the SPARK 2.0, on the other hand it displays the graphic screen. This two features are handled on two different monitors. HCP running in the background prompts necessary messages on a second page of the input/output screen. File handling of the processor is reduced to block transfer of data from and to files given by a number. The mapping of file numbers to physical files on disk is handled by HCP. Three types of data are allowed: 64–bit floating point numbers, 32–bit integers and 64–bit dump–format (e. g. instructions). For a more convenient use both ascii and binary files are supported. Occuring file errors caused by reads over the end or wrong type of data are not passed to SPARK 2.0. Such errors should be solved on the HOST (if possible).

Some special features of HCP left to mention here are e. g. the possibilty to send an interrupt request (maskable or unmaskable) to the SPARK 2.0 at any time, so he enters the coressponding service routine yielding in a communication routine whatever the application might do. While SPARK 2.0 is waiting for a service by the HOST it is possible to peek and poke any memory location or register of the SPARK 2.0. All commands not recognized by HCP are passed to the operating system on the HOST.

3.3 Extended Software

The following set of trigonometric functions has been implemented:

arctan, arctanh, cos, cosh, exp, ln, sin, sinh, sqrt, tan

The used algorithms are taken from [?] and known as CORDIC–Iteration–Algorithms. A drawback of this choice is the slow (linear) convergence of the iterations. But we decided in

favour of the advantages: they are easy to implement, they cover a wide range of functions and they fit into the ROM–part of BM (including the constant tables). All functions get a vector and its length as operands. The longest vector they can deal with has 256 elements. The operand vector is passed through FM and the calculated result is stored there too.

Another extension of the operating system we have implemented is a small graphic packet. An image size of 800 × 600 pixel is mapped into MM. Each pixel has one byte color information which determines a location in the HOSTs color–look–up table. Some of the included functions are e.g. `set_pixel`, `draw_line` and `put_image` (to the HOST). The graphic packet must be installed using an HCP command if it is required by an application program. Its code is loaded into the first part of BM. This mechanism allows for easy modifications and extensions. A call to a graphic function is similar to a system call: parameters and function number are passed through registers in XM, system code is entered through a `call`–instruction.

4 VectorPASCAL Compiler

4.1 General Remarks

We have implemented a compiler for the SPARK 2.0 processor. The main goal of this implementation was to show that the architecture of the SPARK 2.0 can be efficiently used by a compiler. The source language for the compiler is VectorPASCAL which is described in the next section. VectorPASCAL is based on PASCAL and has additional instructions for vector manipulation. Vector instructions are translated into SPARK 2.0 machine instructions which use the FPU without causing bubbles in the pipeline. All other instructions that use the FPU are computed in scalar mode. That means that the result of an operation cannot be used before d machine cycles where d is the depth of the pipeline.

Another possibility would be to use a scalar programming language and a vectorizer which tries to form vector instructions from the loops of the source program. The advantage of this solution would be that existing programs written in the scalar programming language could be used. This is important if someone has spent a lot of time implementing scalar programs and cannot afford to rewrite them. A disadvantage is the difficulty for the vectorizer to find all possible vectorizations. On the other hand the programmer knows best how to vectorize the program and compact vector operations are powerful and expressive means.

The front–end of the compiler translates a program written in VectorPASCAL in an interme-

is reported and the program execution stops. When an operation is applied to a vector v of length L and a scalar s, each element of v is combined with s resulting in a vector of length L. The assignment to a vector

$$v := E(v_1, \cdots, v_k)$$

(where E is an expression with operand vectors $v_1, \cdots, v_k$) is a simultaneous assignment: the assignment to v is executed after the expression E is evaluated completely. This semantic is important, if v is one of the operands of E.

There are two possibilities to build vectors: the specification of a *range expression* and the specification of an expression with an *index vector*. A range expression is given by

< start expression> : (<step expression>) <end expression>

and allows to access elements of an array with a constant stride.

example 1: if x is declared as

x : array [1··10000] of real;

then x[1000:(-2)100] accesses all elements of x with an even index between 100 and 1000 in descending order.

An index vector may be any array of type integer. An expression containing both an index vector and scalars may be used as an index expression too. There is a new operator || ("*concatenate*") to build index vectors. || may be applied to index vectors, scalars and range expressions. The result is a new index vector that contains the indices represented by the operands in the specified order. To fill an index vector with indices of a constant value, a *repeat–expression* can be used as an operand of ||. Such a repeat–expression has the form

$$< \text{expr} > [< \text{repeat–expr} >] \qquad (1)$$

If <repeat–expr> has the value v then (1) represents v integers each having the value of <expr>.

example 2: if ix is declared as

ix : array [1··1000] of integer;

then

ix := ix || 1 : (2) 100 || 1[100];

appends all odd indices between 1 and 100 to the indices already stored in ix. Then it appends 100 indices each having value 1. Note that an expression with a ||-operator must be assigned to an unsubscripted index vector.

We can use a subscripted or an unsubscripted index vector at index position.

example 3: Let x and ix be declared as in examples 1 and 2. The expression

means that the first 100 elements of ix are used as index for array x. The programmer must guarantee that these elements contain indices between 1 and 10000. The expression

x[ix]

means that the occupied part of index vector ix is used as index for x. How is the occupied part of an index vector determined? To answer this question we reserve an additional element (the *length element*) for each integer array ix which contains the number of indices stored in ix. The programmer has no direct access to the length element[1]. The length element is updated each time when ix occurs without a subscript expression on the left hand side of an assignment (or as an argument of a read operation)

$$ix := E(ix_1, \cdots, ix_k)$$

$ix_1, \cdots, ix_k$ are the index vectors used by E. E is usually an expression with || as operator. Let $o_1, \cdots, o_n$ be the operands of E combined by ||. Each o_i represents a certain number of values $n(o_i)$. We get the new value of the length element of ix by adding $\sum_{i=1}^{n} n(o_i)$. The occurence of a subscripted integer array on the left hand side of an assignment does not affect its length element. Hence, if the programmer wants to collect indices in an index vector ix for a later use at index position, he must assign the collected index expressions to the unsubscripted ix. Note that the rule using vectors of equal length both on the left and right hand side of an assignment does not need to be satisfied, if an unsubscripted integer array ix is used on the left hand side of the assignment. But the generated vector should not exceed the declared length of ix.

If an array x is accessed with an index vector ix then ix is kept in XM and the address register of the FPU is loaded in each machine cycle with the indices stored in ix. If ix does not fit in XM, we have to split it in several parts and compute these parts one after another.

[1] But there is a standard function *indexlength* to read the length element.

Sometimes we cannot determine at compile–time how many indices there are in an index vector at run–time. In these cases the generated code must guarantee that a splitting at run–time is performed[2].

When a vector operation contains an operand x which is accessed with an index vector ix, we have two possibilities to compute the operation :

(1) We make a scattered load controlled by ix and load the elements of x which are used in the operation.

(2) We load the entire vector x and we control the operation by ix accessing all elements of x which are used.

(1) is better if the access is sparse, (b) is better when the access is dense, so that elements of x are accessed more than once.

The current implementation of the VectorPASCAL–compiler includes the described vector extensions. Moreover the trigonometric functions described in section 3.3 and the functions of the graphic packet can be used. On the other hand we have suspended the implementation of some control structures (there is no repeat statement, no case statement and no with statement) and of some data types (there is no character type, no set type, no subrange type and no record type) to reduce the development time for the prototype of the compiler. The most severe suspension is that we have neither implemented the dynamic allocation of storage via the new statement nor the pointer type. The justification for these suspensions is that the control statements not implemented can be simulated by other control statements and that the data types not implemented are rarely used in scientific programs. However, the implementation of the missing PASCAL features is straightforward.

5 Run–time Storage Administration

5.1 Run–time Stack

The SPARK 2.0–processor has three memories which can be used to implement the run–time stack: MM, XM and FM. Usually the run–time stack is implemented in MM. But this has several

[2]The same holds for range expressions, if the start– or the end–expression is not known at compile–time.

drawbacks for the SPARK 2.0–processor: MM can only be accessed via the address register MAC. Beyond that no other unit can access data in MM directly. The data must be copied by a MOVE–operation to the local storage of the unit. If the run–time stack was implemented in MM, all variables would be in MM and such simple operations like

$$i := i + 1$$

would require a lot of computation time: MAC is loaded with the address of i, a DMOVE–operation is carried out, the value of i is incremented and finally a DMOVE–operation back to MM is executed. Moreover, by doing so, the FM and XM are hardly used.

Therefore we decided to spread the run–time stack over three memories: MM, XM and FM. XM can be accessed via two address registers XAC and XAC2. Moreover there are two base registers XBASE0 and XBASE1. Each of them allows for a direct access to a block of 16 registers by specifying their address relatively to XBASE0 or XBASE1. This gives us the possibility to access 32 registers of XM very fast — without preloading an address register. The same mechanism is available for FM via the base registers FBASE0 and FBASE1. We will use it to access parts of the run–time stack stored in XM and FM.

-How to split the run–time stack? To answer this question we have to consider the sizes of the memories and their usage for other purposes: The SPARK 2.0–processor is designed for scientific applications, so a typical program will execute many floating–point operations. The operands for those operations should be in FM. Because the source language allows for the computation of vectors, the operands in FM often will be vectors and therefore a large part of FM is needed to store them. We should use only a small part of FM as run–time stack. XM is used to store index vectors, too. But usually only a few of the vector operations use index vectors, so XM will be much less used than FM. Therefore we can use a greater part of XM as run–time stack.

We use the following memory organization: 32 registers of FM contain scalar real variables of the active procedure. We address those registers via FBASE0 and FBASE1. Only one half of them is really used to store variables of the active procedure, the other half is used to evaluate the actual scalar real parameters when a procedure call occurs. The two 16 register parts are used in an alternating manner: Suppose procedure p uses the first part to store its local scalar real variables and parameters. When p calls a procedure q the actual scalar real parameters of

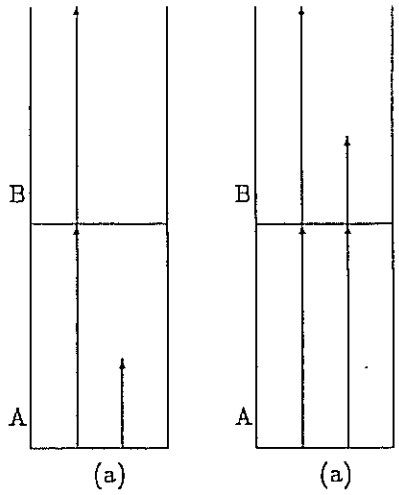

Figure 6: spilling of the XM stack : In (a) the stack contains 64 frames and a procedure call occurs. Part A is spilled and the new stack frame is stored in A. In (b) the stack contains 96 frames (32 of them are spilled) and a procedure call is executed. Part B is spilled and the new stack frame is stored in B.

q to be placed in FM are evaluated (eventually using scalar real variables of p) and stored in the second part. After the actual parameters of q have been evaluated, we save the part used by p in MM and restore it again when q is left. To abbreviate the notation we will often call the reserved part of FM *FM–stack–frame* (although there is no stack in FM).

We use the lower half of XM to store scalar integer variables and organizing entries. This part of XM is organized as a stack consisting of frames each containing 32 entries. This stack grows towards higher addresses and contains one frame for each pendant procedure. The frame of the active procedure is addressed via XBASE0 and XBASE1. A memory map is given in figure 9 and is explained below. The reserved part of XM can hold 64 frames. But this may not be enough, e.g. if a recursive procedure is evaluated. In that case we spill a part of the XM stack into MM. To organize the spilling we split the XM stack into two parts of equal size. To reduce the overhead we spill a whole part containing 32 frames instead of a single frame. If the XM stack contains 64 frames and a procedure call is executed, we spill that part of the XM stack that contains the oldest stack frames and put the new frame in this part (see figure 6). If the XM stack contains 32 frames and a procedure return is executed, the 32 frames spilled last are reloaded again. For this spilling policy to be inefficient the following event must happen: there are 63 frames in the XM stack and two frames are repeatedly pushed and popped. Then after two push operations a spill is executed and after two pop operations the spilled part is reloaded again. But the probability for this to occur in real programs is very low.

In an XM stack frame only 32 entries can be stored. Because this is often not enough space

to hold the data for one procedure, we use a stack in MM too. A MM stack frame contains all entries which do not fit in the XM or FM stack frame. The MM stack grows towards lower MM addresses, the spilled part of the XM stack grows towards higher addresses[3]. This leads to the memory map in figure 7.

To facilitate the spilling of XM frames, each XM frame contains in XBASE0:13 a pointer to the top of the spilled XM stack. Note that the pointer of the topmost frame must also be actualized before a spill is executed. To check whether a spilling is necessary, XBASE0:14 contains a counter representing the number of frames presently occupied in XM. To identify the part of XM that should be spilled, the pointer is negative, if part B is to be spilled next, or positive, if part A is to be spilled next (see figure 6). XBASE0:15 contains the return address of the calling procedure.

We use two pointers to access the MM stack: XBASE1:14 contains the frame pointer FP to access the variables in the actual MM frame. The position of FP can be seen in figure 8. XBASE1:15 contains the stack pointer SP which points to the topmost memory location of the MM stack. It is used to create a new MM frame when a procedure call is executed.

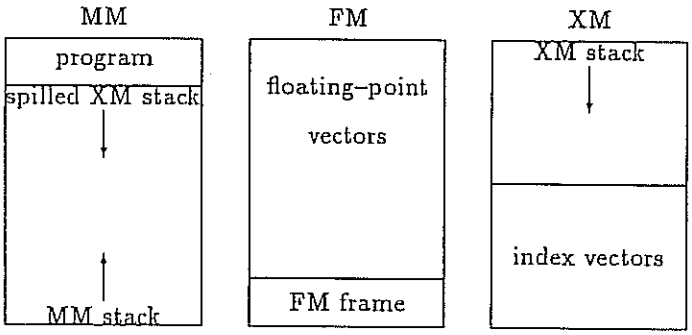

Figure 7: splitting of the run–time stack between MM, XM and FM

To access global[4] variables we use a display mechanism: the display of the active procedure contains a pointer to the topmost stack frame of each statically surrounding procedure. Because

[3]This organization is only possible, because we have no heap. If a heap is used, we must merge the spilled XM stack and the MM stack. The spilled parts of the XM stack are linked together and are stored as parts of the MM stack frames.

[4]If a procedure p accesses a variable x of a statically surrounding procedure q, then x is called global to p.

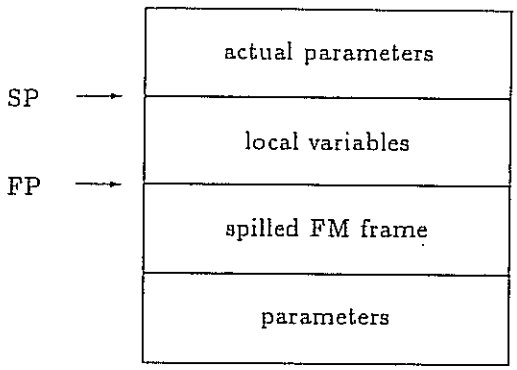

Figure 8: MM stack frame

we have two stacks, we must have two displays. We store the displays in the XM stack frame of the active procedure and restrict the size to 13 entries for each display[5]. Let p_1 be a procedure with nesting level l_1 which uses a variable x of a procedure p_2 with nesting level $l_2 < l_1$. If x is stored in an XM frame we can access x by

$$(\text{XBASE0} : l_2) + ra_{XM}(x)$$

where $ra_{\text{XM}}(x)$ is the relative address of x in the XM stack frame of p_2. If x is stored in a MM frame we can access x by

$$(\text{XBASE1} : l_2) + ra_{\text{MM}}(x)$$

where $ra_{\text{MM}}(x)$ is the relative address of x in the MM stack frame of p_2. If x had been stored in the FM stack frame when p_2 was active, then x has been saved in the MM stack frame of p_2 and we can access it there (with the same relative address than it has in the FM -frame, see figure 8) via

$$(\text{XBASE1} : l_2) + ra_{\text{FM}}(x)$$

When a procedure p_1 calls a procedure p_3 with nesting level l_3 then we must create a new XM stack frame for p_3 with new displays. We get the XM display of p_3 by copying the first l_3 entries of the XM display of p_1 and pushing the new value of XBASE0, which is $(\text{XBASE0}_{old} + 32)$ mod 2048. We get the MM display of p_3 by copying the first l_3 entries of the MM display of

[5]This allows for a nesting level of 12 which is hardly reached in real programs. If a program exceeds nesting level of 12, the compiler reports an error and the programmer has to restructure the program. We suppose the main program to have nesting level 0.

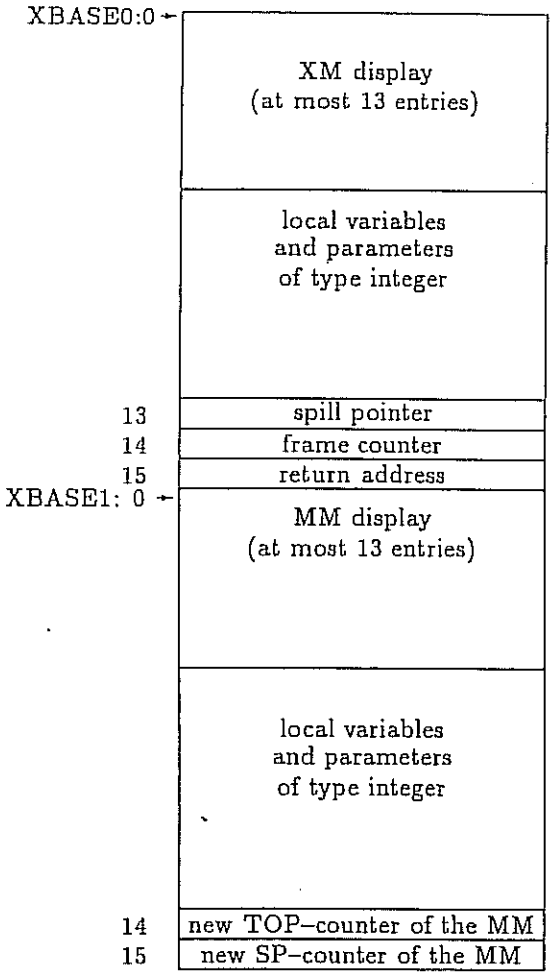

Figure 9: XM stack frame

p_1 and pushing the new value of the MM frame pointer FP. If we compute this value after the actual parameters have been pushed onto the MM–stack, we get this value easily as SP + 16.

Consider the following situation : The XM stack contains 64 frames and p_1 calls p_3, so a part of the XM stack must be spilled. When p_3 is left the XM frames spilled when entering p_3 remain spilled until p_1 is left too. To guarantee that p_1 can access to all variables global to p_1, we must change the XM display of p_1 before updating the XM display for p_3: we have to replace all pointers to XM frames just spilled by pointers in MM to the spilled frame[6]. Note that no display in any other frame is to be changed.

[6] We could avoid the change in the XM display for p_1 when the XM frames spilled when entering p_3 would be restored when leaving p_3 but this would increase the probability of repeated spill and reload.

To mark the difference, the pointers to the spilled XM frames get a negative value. When a procedure p accesses a variable x global to it (with nesting level l) which is stored in an XM frame, we must check the display entry: If the entry is positive, we load XAC with

$$(\text{XBASE0} : l) + ra_{XM}(x)$$

and access the variable. If the entry is negative, we load MAC with

$$(-\text{XBASE0} : l) + ra_{XM}(x)$$

perform a DMOVE–operation to XM before carrying out the specified operation. If x gets a new value, it is restored in MM afterwards.

Usually the displays do not contain 13 entries. So we can use the rest of the XM frame to store scalar local variables and parameters of type integer. These are typically control variables for loops and variables to access array elements. So we get the organization given in figure 9.

The splitting of the run–time stack causes a problem with var–parameters: The var–parameter specifies a memory address, but for what memory? To avoid the reservation of more than 1 memory location, we decode the memory and the address as follows:

- if the formal parameter is a vector, the address references to MM
- if the formal parameter is a scalar of type integer: if the address is negative, then it references to MM, else to XM
- if the formal parameter is a scalar of type real: if the address is negative, then it references to MM, else to FM

The spilling of the XM frames causes another problem. Consider the following situation: while XM contains 63 frames, a procedure p with a formal var–parameter x of type integer is called and the actual parameter is a global variable a stored in part A of the XM stack. When p calls another procedure q, part A of the XM stack is spilled. When q is left, part A remains spilled until p is left, so p cannot access x because a is in the MM. There are three possibilities to solve this problem:

(1) The compiler determines which scalar variables of type integer are used as global actual var–parameter in a procedure call and stores these variables in MM instead in XM. This has the disadvantage of a slower access.

(2) If a spilling of XM frames occurs and the calling procedure has scalar var–parameters of type integer, these are checked: If they reference to a XM location XM(i) that is spilled afterwards, then the address is changed to the MM location of XM(i) after the spill. Note that only the var–parameters of the calling procedure must be checked and that nothing must be done when reloading the spilled XM frames. This solution has the disadvantage of an additional overhead for procedure calls.

(3) We give scalar var–parameters of type integer a copy–restore semantic: The value of the actual parameter is copied when entering the called procedure and the (modified) value is restored when exiting the procedure. Note that this may have a different semantic than the call–by–reference semantic usually used. This is shown by the following example:

```
procedure p (· · · );
   var a : integer;
   procedure q (var x : integer);
   begin
     x:= 1;                (1)
     write(a);
   end
begin
   a:= 0;
   q(a)
end
```

Using copy–restore semantic, statement (1) changes the value of a copy of a and the original value 0 is printed. Using call–by–reference semantic, statement (1) changes the original value of a to 1 and 1 is printed. This solution causes no additional overhead but may yield to other results than the programmer may expect.

Entering a procedure of nesting level l takes 59 + 2l machine cycles, if no XM frames are spilled. Returning from a procedure takes 33 machine cycles, if no reloading of XM frames is necessary. Note that 16 of these machine cycles are necessary to spill or reload the FM frame[7].

[7]This time can be reduced by only spilling and reloading the elements of the FM frame that are really used

Procedure calls seems to be quite expensive, but the choosen memory organization allows a very fast access to local and global data. In scientific programs procedure calls are rare, but data access occur very often, so we expect that the chosen memory organization will be profitable.

5.2 Management of BM

The BM of the SPARK 2.0 can hold only 16K instructions. 4K of these are reserved for the operating system, the run–time storage administration routines etc. 12K are available for user programs. Large programs do not fit into BM, so we have to split these programs and load that part which is going to be executed from MM. One possibility to arrange the loading would be a run–time system as part of the operating system that checks the next instruction to be executed. If this instruction is in a program part that is not in BM the system loads this part. For a conditional branch–instruction this may be the following instruction or the instruction after the specified label. To determine this, the condition must be evaluated.

The disadvantage of such a run–time system is the additional overhead. After the execution of each instruction the following one must be checked. Mostly this instruction will be in BM and no loading is executed – the check was in vain. We think a better solution is to specify the LOAD–instructions by the compiler because it knows best when a load is necessary.

The compiler splits the assembly program in segments $S_0, \cdots, S_{n-1}$ each having at most size 4K, so there fit three segments in BM. Segment S_i gets MM address $4096 \cdot i$ and BM address $(4096 \cdot (i+1)) \bmod 12218$. Note that a segment is always loaded at the same BM position. The segment containing the starting point of the main program is loaded into BM before program execution begins. The compiler uses two types of branch instructions: short and long branches. A short branch is a branch within a segment, a long branch is a branch to another segment. At the end of each segment a long branch to the next segment is inserted. A short branch consists only of one assembly instruction. A long instruction consists of four assembly instructions. These use 6 reserved XM locations. Three of these locations contain the MM address of the segments in BM. The other three locations contain the MM address and the BM address of the target segment and the label of the target instruction. A long branch loads these three entries and jumps to a short subroutine (this branch has the same condition as the branch instruction

to be executed). The subroutine checks the target segment by comparing its MM address with the MM address of the segments in BM. If the target segment is not within BM, it is loaded from MM. Then a branch to the target instruction is executed. Note that this branch has no condition. If the target segment already has been loaded, the subroutine executes between 4 and 8 instructions, depending on the position of the target segment in BM.

Running a program, most of the jumps will be short jumps for which no overhead is necessary at all. When a loop spreads over more than one segment, a long jump is used to jump back to the loop condition (or from the loop condition back to the beginning of the loop, depending on the translation of loops). If the loop does not exceed three segments, no unnecessary loading is performed because the segments containing the loop remain in BM.

The overhead of a long jump takes 7 to 11 instructions. This is a small overhead. Moreover it occurs rarely because most of the jumps executed are short jumps.

References

[APB87] D. Auerbach, W. Paul, A. Bakker, C. Lutz, W. Rudge, F. Abraham : *A Special Purpose Parallel Computer for Molecular Dynamics,* Journal of Physical Chemistry, 1987, 91, 4881

[Fo89] A. Formella : *Entwurf, Bau und Test eines Vektor–Prozessors mit parallel arbeitenden Operationseinheiten,* SPARK 2.0 *Teil I,* Diplomarbeit, Universität des Saarlandes, FB14 Informatik, 1989

[Fo90] A. Formella : *SCP and HCP for* SPARK 2.0, Universität des Saarlandes, FB14 Informatik, unpublished, 1990

[KPR91] C.W. Kessler, W.J. Paul, T. Rauber : *A randomized heuristic approach to register allocation,* to appear in the proceedings of PLILP91, Springer Lecture Notes, 1991

[Lu87] C. Lutz : *SCP and HCP for* SPARK 1.0, IBM Almaden Research Center, unpublished, 1987

[Ob89] A. Obé : *Entwurf, Bau und Test eines Vektor–Prozessors mit parallel arbeitenden Operationseinheiten,* SPARK 2.0 *Teil III,* Diplomarbeit, Universität des Saarlandes, FB14 Informatik, 1989

[Pa82] G. Paul : *VECTRAN and the Proposed Vector/Array Extensions to ANSI FORTRAN for Scientific and Engineering Computation,* Proceedings of the IBM conference on Parallel Computers and Scientific Computation, 1982

[PCM83] R. Perrott, D. Crookes, P. Milligan : *The Programming Language ACTUS,* Software-Practice and Experience Vol. 13, 1983

[PeZa86] R. Perrott, A. Zarea–Aliabadi : *Supercomputer Languages,* Computing Survey, Vol. 18, No. 1, 1986

[Ra90] T. Rauber : *Ein Compiler für Vektorrechner mit optimaler Auswertung von vektoriellen Ausdrucksbäumen,* PhD thesis 1990, Universität des Saarlandes

[Ra90a] T. Rauber : *An Optimizing Compiler for Vector Processors,* Proc. ISMM International Conference on Parallel and Distributed Computing and Systems, 1990

[Ra90b] T. Rauber : *Optimal Evaluation of Vector Expression Trees,* 5^{th} Jerusalem Conference on Information Technology, 1990

[Sch89] D. Schmidt : *Entwurf, Bau und Test eines Vektor–Prozessors mit parallel arbeitenden Operationseinheiten,* SPARK 2.0 *Teil II,* Diplomarbeit, Universität des Saarlandes, FB14 Informatik, 1989