GNU COMPILER COLLECTION BACKEND PORT FOR THE INTEGRAL PARALLEL ARCHITECTURE

Radu HOBINCU[1], Valeriu CODREANU[2], Lucian PETRICĂ[3]

Lucrarea de faţă prezintă procesul de portare a compilatorului GCC oferit de către Free Software Foundation pentru arhitectura hibridă Integral Parallel Architecture, constituită dintr-un controller multithreading şi o maşina vectorială SIMD. Este bine cunoscut faptul că motivul principal pentru care maşinile hibride ca şi cele vectoriale sunt dificil de utilizat eficient, este programabilitatea. În această lucrare vom demonstra că folosind un compilator open-source şi facilităţile de care acesta dispune, putem uşura procesul de dezvoltare software pentru aplicaţii complexe.

This paper presents the process of porting the GCC compiler offered by the Free Software Foundation, for the hybrid Integral Parallel Architecture composed of an interleaved multithreading controller and a vectorial SIMD machine. It is well known that the main reason for which hybrid and vectorial machines are difficult to use efficiently, is programmability. In this paper we well show that by using an open-source compiler and the features it provides, we can ease the software developing process for complex applications.

Keywords: integral parallel architecture, multithreading, interleaved multithreading, bubble-free embedded architecture for multithreading, compiler, GCC, backend port

1. Introduction

The development of hardware technology in the last decades has required the programmers to offer support for the new features and performances of the last generation processors. This support comes as more complex compilers that have to use the machines' capabilities at their best, and more complex operating systems that need to meet the users' demand for speed, flexibility and accessibility.

[1] Eng., Faculty of Electronics, Telecommunications and Information Technology, University Politehnica of Bucharest, Romania, e-mail: radu.hobincu@arh.pub.ro

[2] Eng., Faculty of Electronics, Telecommunications and Information Technology, University Politehnicof Bucharest, Romania, e-mail: vali.codreanu@arh.pub.ro

[3] Eng., Faculty of Electronics, Telecommunications and Information Technology, University Politehnica of Bucharest, Romania, e-mail: lucian.petrica@arh.pub.ro

The problem, as always, is that the more the software grows, the more complex it becomes, and harder to control. One of the ways to fight the growing complexity is the use of abstract high level languages with which you can describe complex behavior by using simple syntax. Compilers for these languages however, are inherently complex and you cannot always use a compiler to compile a compiler. As a programmer for such software, you need intimate knowledge of the architecture of the machine you are working with and experience with low level programming. This is the reason why compilers are the most difficult software systems to implement, and the most difficult to optimize.

The aim of this paper is to detail the porting process of the retargetable GCC compiler suite for a heterogeneous system composed of an interleaved multithreading controller and a SIMD (Single Instruction Multiple Data) engine. The ported compiler can and has been used for writing various kernels in the C high level language.

2. Compilers and GCC

The role of a compiler is to translate the high level syntax of the language it has been designed for into assembler code for a specific target machine. Good compilers also analyze the input code and try to optimize certain constructs and map them as best as possible to the machine's capabilities. This is usually done in three steps:

- Translation of the High Level Language in an internal tree representation;
- Optimization of the tree structure;
- Translation of the tree in assembly language

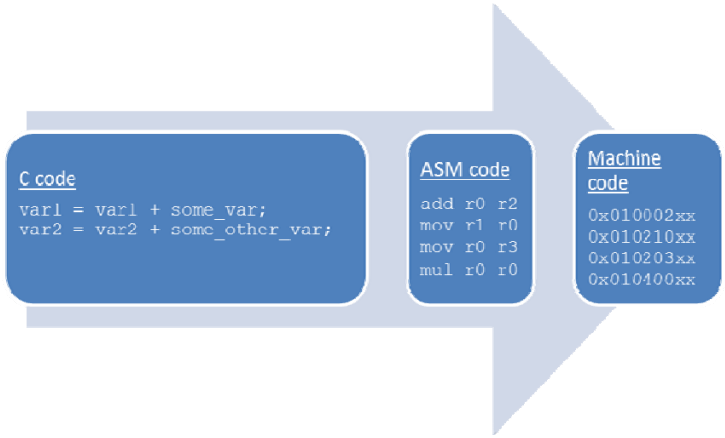

Fig. 1. Compilation flow example

After the assembly code is generated, an assembler is used to translate it into machine code and save it in an object file (.o in Linux or .dll in Windows). Several object files can be merged together by a linker in order to generate a statically linked executable [1].

Compilers, like all other software nowadays come in two flavors: proprietary and open-source. Each major processor manufacturer has developed a set of tools for its own architecture. These mainly consist of an assembler, a compiler and a simulator. Clients are often charged for these tools, as the producer offers support and consultants to help integrate the client's application with the developing environment. Open-source software on the other hand, comes with no support other than public forums and mailing lists, which is the reason why companies prefer the much more costly proprietary version.

The performance of a compiler is given by the level of optimization it is able to make on the code and the capacity to use all the available machine instructions in order to translate the HLL code in the most optimal way.

The GNU Compiler Collection (usually shortened to GCC) is a compiler system produced by the GNU Project supporting various programming languages. GCC is a key component of the GNU toolchain. As well as being the official compiler of the unfinished GNU operating system, GCC has been adopted as the standard compiler by most other modern Unix-like computer operating systems, including GNU/Linux, the BSD family and Mac OS X. GCC has been ported to a wide variety of processor architectures, and is widely deployed as a tool in commercial, proprietary and closed source software development environments. GCC is also available for most embedded platforms, for example Symbian, AMCC and Freescale Power Architecture-based chips. The compiler can target a wide variety of platforms, including videogame consoles such as the PlayStation 2 and Dreamcast. Several companies make a business out of supplying and supporting GCC ports to various platforms, and chip manufacturers today consider a GCC port almost essential to the success of any architecture [2].

In order to port GCC to a different architecture, this architecture needs to be defined. The definition is made up of a header file and a machine description (.md) file. The header file contains macro definitions of the machine's properties like endianess, supported data types, size of the register file and register width, addressing modes, etc. The .md file contains the description of the instruction set in a syntax called RTL.

GCC is open-source. It is provided by the Free Software Foundation under the GNU General Public License (GNU GPL).

So, in theory, GCC is a retargetable compiler which can be ported to new architectures with relatively little effort. However, the original project was not intended for vectorial machines and although the developing community is working continuously to add new features to the suite, it is still difficult to

describe architectures that do not meet certain constrains. This will be developed more in the following sections.

3. Integral Parallel Architecture

The architecture used in this paper is being developed both as a commercial product as well as a PhD project in the University "Politehnica" of Bucharest. It has been described in detail in [3] and it can be seen in Fig. 2.

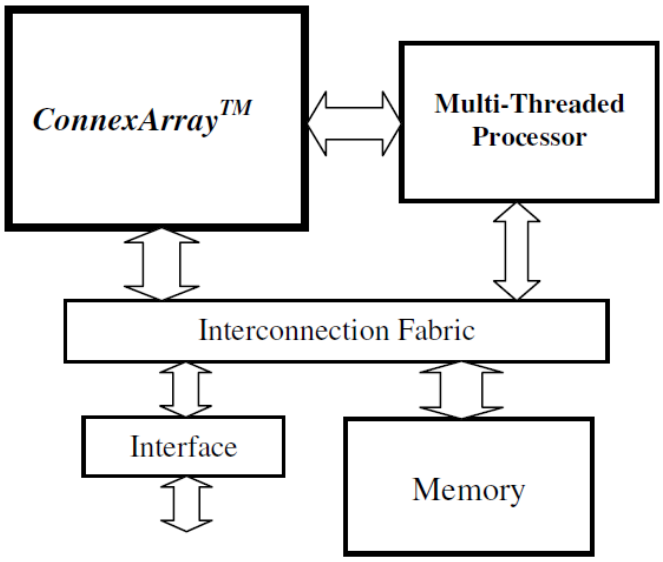

Fig. 2. Integral Parallel Architecture [3]

The system is composed of a BEAM multithreading processor [4] and a vectorial SIMD machine [5]. In order to make the software development easier, the instruction set has been unified for both processors so they can execute largely the same instructions. In order to differentiate between the two execution units, we have created a virtual register files in which the first 16 registers belong to the sequential controller, and the last 16 are vectorial registers.

The first step in starting the actual porting of the compiler backend is to make a list of the machine architecture attributes. These include data types, endianess, register count and size, addressing modes, instruction size, number of pipeline levels, etc. The BEAM controller has the following attributes:

- 16 registers, 32 bits wide, numbered from R_0 to R_{15} (per thread);
- 1 32-bit extension register for mul/div operations (per thread);
- 1 32-bit read-only program counter (per thread);

- Integer only ALU (no floating point operations currently supported);
- Dedicated load/store instructions with 32-bit, 16-bit and 8-bit data access and little endian convention (register indirect);
- Dedicated indexed load/store 32-bit word instructions for stack manipulation;
- 32-bit fixed size RISC type instruction set;
- 2 levels of pipeline (fetch-decode and read-execute-writeback).
- 512MB of byte-access addressable RAM memory space mapped from 0x00000000 to 0x1FFFFFFF (29 effective bits);
- 192MB of byte-access addressable peripheral space mapped from 0x20000000 to 0x2BFFFFFF.

The ConnexArray SIMD engine has the following attributes:

- 128 processing elements (also called cells in this paper), highly scalable;
- 16 vectorial registers, 16-bit wide, numbered from R16 to R32 (per cell);
- 1 16-bit extension register for mul/div operations (per cell);
- 1 16-bit shift register with left-right connections in order to transfer data between cells (per cell).
- Integer only ALU (no floating point operations supported) (per cell);
- Dedicated load/store instructions for accessing the local vectorial memory;
- 32-bit fixed size RISC type instruction set;
- 1024 of 16-bit access addressable local RAM memory space mapped from 0x00000000 to 0x00000C00 (11 effective bits, last 2 bits discarded);
- Distribution network for loading scalar values into vectors;
- Reduction network ($\log_2 128$ latency) for computing scalar results from vectors (possible operations are SUM, MAX, and Boolean OR);
- Boolean engine for selecting/ masking cells for conditional operations.

The instructions formats for the BEAM controller and ConnexArray are presented in Fig. 3.1 and the formats for the Boolean section in the SIMD engine are presented in Fig. 3.2.

Value	0	opCode	boolInsn	right	left	dest
Bit Size	1	8	8	5	5	5

Value	1	opCode	immediate	left	dest
Bit Size	1	5	16	5	5

Fig. 3.1 – BEAM Controller Instruction Formats

Value	0000	boolOpCode
Bit Size	4	4

Value	boolOpCode	flagIndex
Bit Size	4	4

Fig. 3.2. Boolean Instruction Formats

4. GCC Backend Development

We will call the new architecture for GCC "connex". In order to be able to build a C compiler for a specific architecture, entries for that architecture need to be added in the config.sub and gcc/config.gcc files as follows:

- In connex.sub, under "`$basic_machine`" at line 143, "`connex`" architecture name must be added, and also "`connex*-*`" in order to also match the architecture with company name;
- In gcc/config.gcc, at line 265, under "`case ${target} in`", the following entry must be added:

```
Connex*-*-*)
    cpu_type=connex
    ;;
```

- In gcc/config.gcc, around line 600, under "`case ${target} in`", the following entry must be added:

```
connex*-*-*)
gas=no
gnu_ld=no
tm_file=connex/${target_noncanonical}.h
md_file=connex/${target_noncanonical}.md
out_file=connex/${target_noncanonical}.c
tm_p_file=connex/${target_noncanonical}-
protos.h
            ;;
```

Each port for the GCC suite has a series of description files found in the <root>/gcc/config directory where <root> is the root folder of the GCC

distribution package. The entry above tells the compiler that the definition files for the "connex" target are in <root>/gcc/config/connex directory as follows:

- connex.h – the header file in which all macros and architecture properties are defined;
- connex.md – the machine description file which contains RTL code for instruction pattern matching and generation;
- connex.c – implementations of any functions needed in the header file;
- connex-protos.h – prototypes for public functions in the connex.c file.

In the connex.h file, there are many macros that need to be defined in order for the compiler to work properly. However, we will only describe here the ones that are architecture dependent and have a meaning for our implementation.

Table 1

Macro List Description for Architecture Properties

`#define BYTES_BIG_ENDIAN 0`	This macro tell the compiler that the system memory access is in little endian mode
`#define STACK_BOUNDARY 32`	This is the minimum alignment enforced by hardware for the stack pointer on this machine. Since the 32bits access is just as fast as the byte access, we prefer to keep the stack aligned to 32bits as well.
`#define SLOW_BYTE_ACCESS 1`	The byte and word accesses are just as fast for this machine, so we define this in order to tell the compiler that the byte access is slow and he should use word access when possible.
`#define FIRST_PSEUDO_REGISTER 35`	The first pseudo register is the first index after all the hardware registers. So we have 16 scalar registers, 16 vectorial registers, 1 scalar extension register, 1 vectorial extension register and 1 shift register that make out 35 total registers (from 0 to 34).

`#define FIXED_REGISTERS \`
`{0,0,0,0, \`
`0,0,0,0, \`
`1,1,1,0, \`
`0,1,1,1, \`
`0,0,0,0, \`
`0,0,0,0, \`
`0,0,0,0, \`
`0,0,0,0, \`
`1,1,1}`
	The fixed registers are the ones with special functions during the program execution and are not allocated for general computation. In our case, register R8 is dedicated for protected mode return address from a software interrupt function, R9 is dedicated for protected mode return address from a hardware interrupt function, register R10 is the stack pointer during protected mode kernel execution. R13 and R14 are the frame, and respectively the stack pointer in user mode and the register R15 is the link register in user mode. Registers R32, R33 and R34 are the scalar extension register, vectorial extension register and

	shift register.
`#define STACK_POINTER_REGNUM 14` `#define HARD_FRAME_POINTER_REGNUM 13`	Definition for the stack and frame pointer.
`#define FUNCTION_VALUE_REGNO_P(REGN) ((REGN) == 12 || (REGN) == 31)`	Definition for the return value registers (returns 1 if the REGN argument is valid for returning a value.

The contents of the connex.c file are too large to go into but we need to point out that this C source code contains functions needed for prologue and epilogue operations during calls and returns, maps data types to registers by telling the compiler how many registers are needed to hold a particular data type, which data types can a particular register hold, how many registers must be saved for a certain call instruction, which address constructs are valid and can be used as addresses and just as important, how to output the assembly code.

The machine description file contains RTL code for pattern matching and generation. Relevant examples from several instruction types can be found in table 2.

Table 2

Machine Description Pattern Examples

`(define_insn "jnz"`
`[(set (pc)`
`(if_then_else (ne (match_operand:SI 0 "register_operand" "r") (const_int 0)) (label_ref (match_operand 1 "" "")) (pc))`
`)]`
`""`
`"jnz %0,%l1")`
	This control instruction defines a pattern for a jump-if-not-zero. It is used for matching the specified pattern and generates the "jnz" assembly instruction.

`(define_expand "call"`
`[(call (match_operand:SI 0 "" "")`
`(match_operand:SI 1 "" ""))`
`(clobber (reg:SI 15))]`
`""`
`"`
`operands[0] = force_reg(SImode, XEXP(operands[0], 0));`
`")`
	This is an interesting example of an expanded pattern. The call instruction takes the argument out of a register and not as an immediate value as most processors do. So this define_expand forces the address argument into a SImode register (Standard Integer).

`(define_insn ""`
`[(set (match_operand:QI 0 "register_operand" "=r,r")`
`(match_operand:QI 1 "nonmemory_operand" "r,I")`
	This is an example of a pattern used for byte constant loading and register moving. Letter "r" stands for general purpose register and "I" is a 16 bits signed integer. This is a

)] "" "@ move %0,%1 vload %0,%c1")	meta-pattern specifying transfers from “r” to “r” (move) and from “I” to “r” (vload).
(define_insn "" [(set (mem:SI (plus:SI (match_operand:SI 0 "register_operand" "r") (match_operand:SI 1 "const_short_operand" "I"))) (match_operand:SI 2 "register_operand" "r"))] "" "iwr %0,%2,%c1")	Pattern for indexed memory writes. It specifies a base register and 16 bits signed constant that is added to the base to form the final address. Also specifies the register for the value to be written.
(define_expand "divmodsi4" [(parallel [(set (match_operand:SI 0 "register_operand" "") (div:SI (match_operand:SI 1 "register_operand" "") (match_operand:SI 2 register_operand" ""))) (set (reg:SI EXT) (mod:SI (match_dup 1) (match_dup 2)))]) (set (match_operand:SI 3 "register_operand" "") (reg:SI EXT)) (use (match_dup 0)) (clobber (reg:SI EXT))] "" "")	This is the expansion for the division and remainder operation. Since the “div” instruction also places the remainder of the operation in the extension register, this pattern tells the compiler that he should then move the result of the modulo operation into its final destination, and that in this process, the extension register is clobbered (overwritten).
(define_insn "subv128hi3" [(set (match_operand:V128HI 0 "register_operand" "=v") (minus:V128HI (match_operand:V128HI 1 "register_operand" "v") (match_operand:V128HI 2 "nonmemory_operand" "v")))] "" "add %0,%1,%2")	This is an example for a vectorial addition pattern. Notice the V128HI mode which translates as a vector of 128 elements of half-integer values.

The most difficult problem while trying to describe a heterogeneous system for the GCC, is its lack of flexible stack support. This is mostly due to the fact that GCC was originally designed for old fashioned serial architectures like

the x86 and all future development has been done as demanded by the evolution of the modern machines which currently only support SIMD operations for narrow data types.

That means that GCC can only be aware of one stack pointer so any argument passing on the stack and any spill/ fill operations needed for any data types are done on that one stack. Unfortunately our system contains two different RAM memories: the general 512MB DDR system memory, and the 1024 vector local memory inside the Connex Array. We were hoping to adapt the stack mechanism in GCC to support two stack pointers, one for scalar variables, and one for vectors, each pointing to addresses in these two different memories. However, since that proved to be difficult, the solution was to define a global variable in one of our library files which maps the entire Array local memory. This allows the programmer full access to that resource but limits the compiler flexibility since stack operations for vectors are not supported.

5. Assembler Inline

Several of the features of the Connex System cannot be defined in the compiler backend. This means that the compiler itself is not aware of all the things the processors can do. In order to give the programmer access to these features, we created macros and functions which manually insert the required instructions in the assembly file. Some examples are shown in table 3.

Table 3

Inline Assembly Examples

`#define compareAndSwap(address, valueToCompare, valueToWrite) \`
`__asm__ __volatile__( \`
`"cas %0,%1,%2" \`
`: "=r" (valueToWrite)\`
`: "r" (address), "r" (valueToCompare) \`
`)`
	This is a description of a compare-and-swap type operation required for mutex implementation. It tells the compiler that it should emit a "cas" assembler instruction in which the first operand is a destination and it corresponds to the "valueToWrite" variable and the last two operands correspond to the variables "address" and "valueToCompare", all stored in general purpose registres.

`#define INDEX \`
`({register vector i; \`
`__asm__ __volatile__("ldix \t%0" \`
`:"=v" (i) \`
`);i;})`
	This macro is used for loading an index vector in a vectorial register. The index vector is a vector in which all cells contain the index of their position (0 in the first cell, 1 in the seconds, 2 in the third, etc.).

6. Compilation Results

After the successful porting of the compiler, we have written several applications for testing purposes with results published in [6]. Even though the generated code is not as efficient as assembly programming, it shows a high level of optimization, especially for the controller instructions. Some examples of compiled code are shown in table 4.

Table 4

Compiled Code Examples

```
int mutex_lock(struct mutex * mtx) {
      int _originalPid =
getSWThreadId();
      int _pid = _originalPid;
      do{
            compareAndSwap(&mtx-
>threadId, 0, _pid);
            //if mtx->threadId == 0,
it means the mutex is free, so lock it
            _pid = _originalPid;

      compareAndSwap(&mtx->threadId,
_originalPid, _pid);            //if
mtx->threadId == _originalPid means
the lock is ours
      }while(_pid != _originalPid);
      mtx->lockCount++;
      return 0;
}
```

```
      .align 1024
      .globl mutex_lock
mutex_lock:
      write R14,R15
      iwr R14,R14,-4
      iwr R14,R13,-8
      move R13,R14
      iwr R13,R0,-12
      iwr R13,R1,-16
      iwr R13,R2,-20
      iwr R13,R3,-24
      iadd R14,R13,-28
      ird R3,R13,4
      vload
R0,HIGH(getSWThreadId)
      vlload
R0,R0,LOW(getSWThreadId)
      call R0
      iadd R0,R3,4
      vload  R2,0
L8:
;# 14 "mutex.c" 1
      cas R1,R0,R2
;# 16 "mutex.c" 1
      cas R1,R0,R12
      eq R1,R12,R1
      jz R1,L8
      read R3,R0
      iadd R0,R0,1
      write R3,R0
      vload  R12,0
      ird R0,R13,-12
      ird R1,R13,-16
      ird R2,R13,-20
      ird R3,R13,-24
      move R14,R13
      ird R13,R14,-8
```

	read R14,R15 ajmp R15

vector *outputFrame = &localMemory[_offset];
vector *inputFrame = &localMemory[FRAME_HEIGHT + _offset];
int i;
register vector _tmp1;
register vector _tmp2;
lock(index);
read(inputFrame, FRAME_HEIGHT/THREADS, inBuffer + _offset * 2 * FRAME_WIDTH);
unlock(index);
for (i = _offset == 0; i < FRAME_HEIGHT/THREADS - (_offset == (FRAME_HEIGHT - FRAME_HEIGHT / THREADS)); i++){
/* apply filter vertically */
outputFrame[i] = bitRightShift(inputFrame[i-1] + bitLeftShift(inputFrame[i],1) + inputFrame[i+1], 2);
}
	ird R1,R13,4
	vload R2,HIGH(localMemory)
	vlload R2,R2,LOW(localMemory)
	read R2,R2
	ishl R3,R1,2
	add R0,R2,R3
	iwr R13,R0,-48
	iadd R0,R1,120
	ishl R0,R0,2
	add R0,R2,R0
	vload R12,0
	jnz R1,L7
	vload R12,1
	add R3,R0,R3
	iadd R1,R0,4
	iwr R13,R1,-52
	move R2,R1
	iadd R1,R0,8
	ird R4,R13,-48
	iadd R6,R4,4
	move R5,R12
	vload R7,2
	L8:
	read R2,R31
	;# 38 "image_filter_mth.c" 1
	shl R31,R31,R5
	read R3,R16
	add R31,R31,R16
	read R1,R16
	add R31,R31,R16
	;# 38 "image_filter_mth.c" 1
	shr R31,R31,R7
	;# 42 "image_filter_mth.c" 1
	lshr R31,R5
	;# 42 "image_filter_mth.c" 1
	shwait
	;# 42 "image_filter_mth.c" 1
	shget R17
	;# 43 "image_filter_mth.c" 1
	rshr R31,R5
	;# 43 "image_filter_mth.c" 1
	shwait
	;# 43 "image_filter_mth.c" 1
	shget R16

	add R31,R31,R31
	add R31,R17,R31
	add R31,R16,R31
	;# 44 "image_filter_mth.c" 1
	shr R31,R31,R7
	write R6,R31
	iadd R12,R12,1
	iadd R3,R3,4
	iadd R2,R2,4
	iadd R1,R1,4
	iadd R6,R6,4
	vload R11,118
	gt R4,R12,R11
	jz R4,L8

The results show that for scalar operands, the compiler generates optimized code. However, due to its inability to handle multiple stacks, the use of inlined assembly instructions results in suboptimal constructs for parallel operands.

7. Conclusions

Although new architectures definitions (back-ends) can be added to the GCC, this is not a straightforward operation. These definitions are done in a non-standard format (the RTL syntax) which needs to be understood before the actual porting takes place. Also, several components of the compiler middle-end (tree parser, optimizer, etc.) make assumptions about the machine which are not always correct (like the maximum width of a SIMD operand). Moreover, the middle and back-end sometimes cooperate in ways that are not described in the documentation and can confuse a person that is not familiar with the actual implementation of the compiler. That means that there are cases in which, for non-standard machine architectures, the porting of the compiler is not limited to the description of the back-end, but also requires the adaptation of the middle-end and perhaps even the adding of additional passes or RTL constructs.

In this current case, the lack of support for wide vectorial and multi-memory machines is the greatest drawback of the GNU Compiler which has been bypassed by using suboptimal inline assembly constructs. Adding said support would require extensive modification of the middle-end and intimate knowledge of the compiler implementation.

In the end, we can conclude that the porting effort of the GCC was required to further develop complex applications for the Connex System in order to prove its superiority in both performance and power consumption over traditional machines. This paper can also be used as a starting point for similar

projects, especially considering the low amount of documentation available for the GCC suite.

REFERENCES

[1] *Stroustrup B.*, "A history of C++: 1979—1991", The second ACM SIGPLAN conference on History of programming languages, 1993

[2] *Stallman R. M*, GNU compiler collection internals. 2002, Vol. Free Software Foundation.

[3] *Bumbacea P., Codreanu V., Hobincu R., Petrica L., Stefan G. M.*, "Technology driven architecture for integral parallel embedded computing", CAS 2010 Proceedings, Vol. 1, pp. 35–42

[4] *Codreanu V., Hobincu R.*, "Performance gain from data and control dependency elimination in embedded processors", 9th International Symposium on Electronics and Telecommunications (ISETC), 2010, pp. 47-50

[5] *Malita M., Stefan G. M., Stoian M.*, "Complex vs. Intensive in Parallel Computation", International Multi-Conference on Computing in the Global Information Technology, 2006, p. 26

[6] *Codreanu V., Petrică L., Hobincu R.*, "Increasing Vector Processor Pipeline Efficiency with a Thread-Interleaved Controller", accepted at the 15th International Conference on System Theory, 2011