C Compiler Design for a Network Processor

J. Wagner, R. Leupers

Abstract— **One important problem in code generation for embedded processors is the design of efficient compilers for target machines with application specific architectures. This paper outlines the design of a C compiler for an industrial application specific processor (ASIP) for telecom applications. The target ASIP is a network processor with special instructions for bit-level access to data registers, which is required for packet oriented communication protocol processing. From a practical viewpoint, we describe the main challenges in exploiting these application specific features in a C compiler, and we show how a compiler backend has been designed that accommodates these features by means of compiler intrinsics and a dedicated register allocator. The compiler is fully operational, and first experimental results indicate that C-level programming of the ASIP leads to good code quality without the need for time-consuming assembly programming.**

I. Introduction

The use of application specific instruction set processors (ASIPs) in embedded system design has become quite common. ASIPs are located between standard "off-the-shelf" programmable processors and custom ASICs. Hence, ASIPs represent the frequently needed compromise between high efficiency of ASICs and low development effort associated with standard processors or cores. While being tailored towards certain application areas, ASIPs still offer programmability and hence high flexibility for debugging or upgrading. Industrial examples for ASIPs are Tensilica's configurable Xtensa RISC processor [1] and the configurable Gepard DSP core from Austria Micro Systems [2].

Like in the case of standard processors, compiler support for ASIPs is very desirable. Compilers are urgently required to avoid time-consuming and error-prone assembly programming of embedded software, so that fast time-to-market and dependability requirements for embedded systems can be met. However, due to the specialized architectures of ASIPs, classical compiler technology is often insufficient, but fully exploiting the processor capabilities demands for more dedicated code generation and optimization techniques.

A number of such code generation techniques, intended to meet the high code quality demands of embedded systems, have already been developed. These include code generation for irregular data paths [3], [4], [5], [6], [7], address code optimization for DSPs [8], [9], [10], [11], and exploitation of multimedia instruction sets [12], [13], [14]. It has been shown experimentally, that such highly machine-specific techniques are a promising approach to generate high-quality machine code, whose quality often comes close to hand-written assembly code. Naturally, this has to be paid with increased compilation times in many cases.

While partially impressive results have been achieved in code optimization for ASIPs in the DSP area, less emphasis has been so far on a new and important class of ASIPs for bit-serial protocol processing, which are called *Network Processors* (NPs). The design of NPs has been motivated by the growing need for new high bandwidth communication equipment in networks (e.g. Internet routers and Ethernet adapters) as well as in telecommunication (e.g. ISDN and xDSL). The corresponding communication protocols mostly employ bit stream oriented data formats. The bit streams consist of *packets* of different length, i.e. there are variable length header packets and (typically longer) payload packets. Typical packet processing requirements include decoding, compression, encryption, or routing.

A major system design problem in this area is that the required high bandwidth leaves only a very short time frame (as low as a few nanoseconds) for processing each bit packet arriving at a network node [15]. Even contemporary high-end programmable processors can hardly keep pace with the required real-time performance, not to mention the issue of computational efficiency, e.g. with respect to power consumption.

There are several approaches in ASIC design that deal with efficient bit-level processing, such as [16], [17], [18], [19]. However, all these solutions require highly application specific hardware. On the other hand, the design of hardwired ASICs is frequently not desirable, due to the high design effort and low flexibility. As a special class of ASIPs, NPs represent a promising solution to this problem, since their instruction sets are tailored towards efficient communication protocol processing. The advantage of this is illustrated in the following.

Since the memories of transmitters and receivers normally show a fixed word length (e.g. 8 or 16 bits), relatively expensive processing may be required on both sides when using standard processors (Fig. 1): At the beginning of a communication the packets to be transmitted are typically aligned at the word boundaries of the transmitter. For storing these words into the send buffer, they have to be packed into the bit stream format required by the network protocol. After transmission over the communication channel, the packets have to be extracted again at the receiver side, so as to align them at the receiver word length, which may even be different from the transmitter word length.

Obviously, this data conversion overhead reduces the benefits of the bit stream oriented protocol. In contrast, NPs may be designed to be capable of directly processing bit packets of variable length, i.e. in the form they are stored in the receive buffer. This feature largely reduces the data transport overhead.

NPs are relatively new on the semiconductor market. There are only a few standard chips (e.g. from Intel and IBM), and several in-house designs (see the overview in [15], which also describes NP development efforts at STMi-

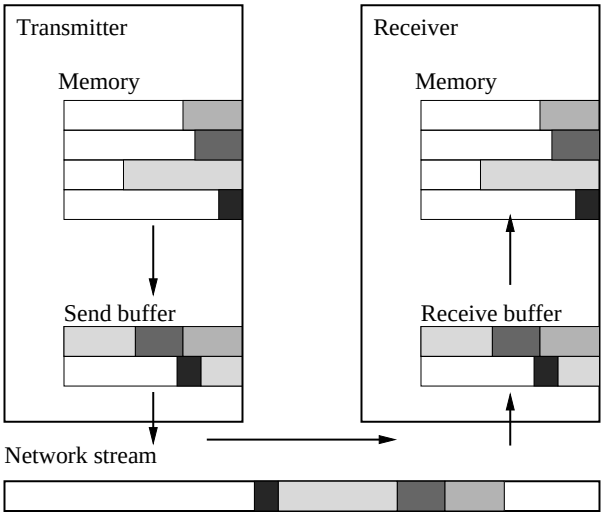

Fig. 1. *Communication via bit stream oriented protocols*

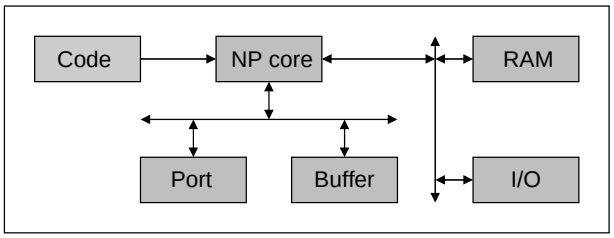

Fig. 2. *Infineon NP architecture*

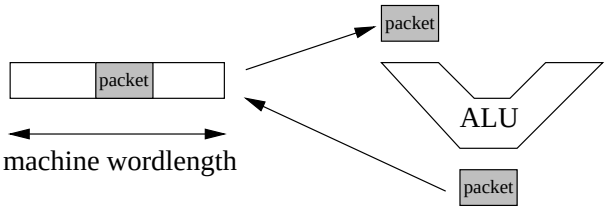

Fig. 3. *Processing of variable length bit packets*

croelectronics). In this paper, we focus on a specific machine, the Infineon Technologies Network Processor [20].

Efficient compiler design for NPs is at least as challenging as for DSPs, since the dedicated bit-packet oriented instructions are not easily generated from a high-level language like C. In contrast to the approach taken in [15], which is based on the retargetable FlexWare tool suite [21], we decided to develop a nearly full custom compiler backend. This is essentially motivated by the need to incorporate C language extensions and a dedicated register allocator, which will become clear later. Another approach related to our work is the Valen-C compiler [22], a retargetable compiler that allows the specification of arbitrary bit widths of C variables. However, there is no direct support for NP applications.

The purpose of this paper is to show how an efficient C compiler for an advanced NP architecture has been implemented, and to describe the required machine-specific code generation techniques. More specifically, we show how bit packet processing is made available to the programmer at the C level, and how the register allocator needs to be designed to handle variable-length bit packets in registers, which is not directly possible by classical techniques.

The remainder of this paper is structured as follows. In section II, the Infineon NP architecture and its instruction set are described in more detail. Section III outlines the problems associated with modeling bit-level processing in the C language. Section IV describes the compiler design, with focus on the backend components. Experimental results are presented in section V. Finally we give conclusions and mention directions for future work in section VI.

II. Target architecture

Fig. 2 shows the overall architecture of our target machine, the Infineon NP [20]. The NP core shows a 16-bit RISC-like basic architecture with 12 general-purpose registers and special extensions for bit-level data access. This principle is illustrated in Fig. 3.

The NP instruction set permits performing ALU computations on bit packets which are not aligned at the processor word length. A packet may be stored in any bit index subrange of a register, and a packet may even span up to two different registers. In this way, protocol processing can be adapted to the required variable packet lengths instead of the fixed machine word length. However, this *packet-level addressing* is only possible within registers, not within memory. Therefore, partial bit streams have to be loaded from memory into registers before processing on the ALU can take place (Fig. 4). The size and position of the different bit fields are statically known from the C source code for each specific application.

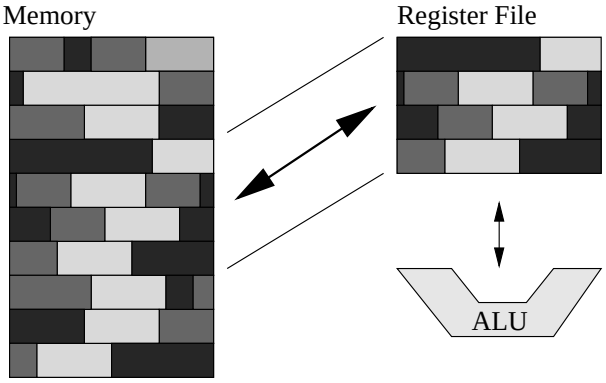

Fig. 4. *Data layout in memory and registers: Bit packets are not aligned at memory or register word lengths*

In order to enable packet-level addressing of unaligned data, the NP instruction set permits the specification of offsets and operand lengths within registers. The general instruction format is as follows:

```
CMD reg1.off, reg2.off, width
```

"CMD" denotes the assembly command, "reg1.off" and "reg2.off" denote argument registers with a corresponding offset, and "width" is the bit width of the operation to be

performed. The use is shown in Fig. 5: Any bit packet is addressed by means of the corresponding register number, its offset within the register, and the packet bit width. If offset plus width are larger than the register word length (16 bits), then the packet spans over two registers (without increasing the access latency, though). Especially this feature is very challenging from a compiler designer's viewpoint. The width of argument and result packets must be identical, and one of the two argument registers is also the result location of any ALU operation. Therefore, two offsets and one width parameter per instruction are sufficient.

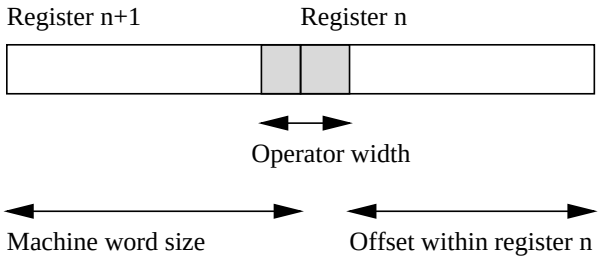

Fig. 5. *Packet-level addressing within registers*

III. Bit packet processing in C

Although possible, the description of bit packet-level addressing in the C language is inconvenient, since it can only be expressed by means of a rather complex shift and masking scheme. The code readability (and thus maintainability) is poor, and furthermore the masking constants might make the code machine-dependent, in case they depend on the processor word length.

A. Use of compiler-known functions

As outlined in section II, the NP instruction set allows to avoid costly shift and mask operations by means of special instructions for packet-level addressing. In the C compiler, bit packet manipulation is made visible to the programmer by means of *compiler-known functions* (CKFs) or *compiler intrinsics*. The compiler maps calls to CKFs not into regular function calls, but into fixed instructions or instruction sequences. Thus, CKFs can be considered as C-level macros without any calling overhead. The CKF approach has also been used in several C compilers for DSPs, e.g. for the Texas Instruments C62xx.

Using CKFs, the programmer still has to have detailed knowledge about the underlying target processor, but readability of the code is improved significantly. In addition, by providing a suitable set of simulation functions for the CKFs, C code written for the NP is no longer machine-dependent but can also be compiled to other host machines for debugging purposes.

We illustrate the use of CKFs with a simple example. Consider a case, where we would like to add the constant 2 to a 7-bit wide packet stored in bits 3 to 9 of some register denoted by the C variable a. In standard C this can only be expressed by means of a complex assignment as follows:

```
a = (a & 0xFE03) | ((a + (2<<2)) & 0x01FC);
```

Even though this expression may still be simplified somewhat by standard compiler optimization techniques, e.g. constant folding, it would translate into a relatively large instruction sequence on a standard processor. In contrast, the NP can implement the entire assignment within a single instruction. For this purpose, we introduce a *packet access* (PA) CKF of the following form:

```
PA(int op, int var1, int off1,
   int var2, int off2, int width);
```

Parameter "op" denotes the operator, "var1" and "var2" are the operands, "off1" and "off2" are the operand packet offsets, and "width" denotes the packet bit width. The CKF directly reflects the packet-level NP instructions illustrated in Fig. 5. The "op" parameter selects the operation (e.g. ADD, SUB, SHIFT, ...) to be performed on the arguments "var1" and "var2". In addition, the required intra-register offsets and the packet bit width are passed to function PA. Using function PA, the above example can be expressed very simply in C as follows:

```
int a, b;
...
b = 2;
PA(PA_ADD, a, 3, b, 0, 7);
```

Parameter PA_ADD (selecting an ADD instruction) is specified as a constant in a C header file. The scalar variables a and b are mapped to registers in the assembly code by the register allocator.

B. Bit packet access in loops

As exemplified above, CKFs provide an elegant way to reduce the description effort of bit packet processing in C. Most urgently, CKFs are required in case of bit packet *arrays*, that are indirectly accessed within loops, so as to avoid unacceptable quality of compiler-generated code.

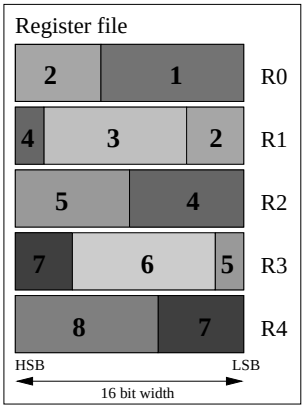

Fig. 6. *Array of bit packets*

Consider the example in Fig. 6, where we have to process an array of 8 packets, each of 10 bit length. The bit packet array in turn is stored in an array of five 16-bit registers (R0 - R4). As a consequence, the bit packets are not aligned at the register word length, and some packets even cross register boundaries.

Suppose, we want to compute the sum over the bit packets within a loop. In standard C, this would require code as shown in Fig. 7. Due to the unaligned bit packets, the register file pointer *elem* must not be incremented in every loop iteration, but only if a bit packet crosses a register boundary. Therefore, control code is required within the loop, which is obviously highly undesirable with respect to code quality.

```
int A[5];
int sum = 0,offset = 0,elem = 0, bp;
for (bp = 1; bp <= 8; bp++)
{
   if(offset+10<=16)
     sum += (A[elem] >> offset) & 0x03ff;
   else
        sum += (A[elem]>>offset)
             | (((A[elem+1]<<(16-offset))& 0x03ff );
   offset += 10;
   if (offset>15) { elem++; offset -= 16; }
}
```

Fig. 7. *Bit packet array access in a loop*

In order to avoid such overhead, the NP instruction set architecture provides means for indirect access to unaligned bit packet arrays via a bit packet pointer register. In the compiler, we again exploit this feature by CKFs. The modified C code with CKFs for the above sum computation example is given in Fig. 8. The array A is declared with the C type attribute *register*. This attribute instructs our compiler backend to assign the whole array to the register file. In contrast, a regular C compiler would store the array (like other complex data structures) in memory. This concept of *register arrays* is required, since the NP machine operations using packet-level addressing only work on registers, not on memory. We assume that register arrays always start a register boundary.

The variables *PR1* and *PR2* are pointers. By being operands of the CKF *INIT* they are introduced to the backend as *pointers to bit packets*. The compiler checks that pointers to bit packets are assigned exclusively within CKFs, since otherwise incorrect code might result. The backend exploits the knowledge about which pointer belongs to which element/array during life time analysis in the register allocation. In the example, *PR2* is used to traverse the different bit packets of array A, while *PR1* constantly points to the *sum* variable.

If the number of physical registers is lower than the number of simultaneously required register variables, spill code will be inserted. The register allocator uses the names of the pointer registers in such a case to identify the register arrays which have to be loaded into the register file for a given indirect bit packet operation. The *INIT* CKF translates to assembly code as the load of a constant into a register. In this constant the name of the register, the offset, and the width of the bit packet where the pointer points to is encoded. In the example from Fig. 8 the pointer *PR2* points to the first element of the bit packet array. The name of the register where the bit packet is located is not known before register allocation. Therefore the backend works on symbolic addresses before register allocation. The symbolic addresses are automatically replaced after the register allocation phase. *PAI(ADD,...)* is the CKF for a single indirect addition, like in the C expression "`*p + *q`". The backend creates a single machine operation for this CKF. In order to keep the number of CKFs low, we specify the arithmetic operation as the first parameter of the CKF instead of having a dedicated CKF for each operation.

The CKF *INC* denotes the increment of a bit packet pointer. Like the increment of a pointer in ANSI C the pointer will point to the next array element after the call to *INC*. Because the NP supports bit packet pointer arithmetic in hardware, this requires again only a single machine instruction, independent of whether or not advancing the pointer requires crossing a register boundary.

Obviously, the source code in the example is specified in a very low level programming style. However, the programmer still gets a significant benefit from use of the compiler. First of all, it permits the use of high level language constructs, e.g. for control code and loops. In addition, address generation and register allocation are performed by the compiler, which keeps the code reusable and saves development time.

```
register int A[5];
int *PR1, *PR2;
...
int sum = 0;
INIT(PR1, sum, 10);
INIT(PR2, A, 10);
for (i = 1; i <= 8; i++)
{
    PAI(ADD, PR1, PR2);
    INC(PR2);
}
```

Fig. 8. *Bit packet array access with CKFs*

IV. Compiler design

Like most other compilers, the NP C compiler is subdivided into a frontend and a backend part. The frontend is responsible for source code analysis, generation of an intermediate representation (IR), and machine-independent optimizations, while the backend maps the machine-independent IR into machine-specific assembly code.

As a frontend, we use the LANCE compiler system developed at the University of Dortmund [28]. LANCE is a machine-independent, optimizing ANSI C frontend. There is no support for automatic retargeting, but LANCE comprises a backend interface that allows for direct coupling between the generated three address code IR and the data flow tree format used by contemporary code generator generator tools.

The C compiler backend is subdivided into code selection and register allocation modules. The code selector maps data flow trees as generated by the LANCE C frontend and its backend interface into NP assembly instructions. As in many compilers for RISCs, during this phase an infinite number of *virtual registers* are assumed, which are later folded to the available amount of *physical registers* by the register allocator.

A. Code selection

The code selector uses the widespread technique of tree pattern matching with dynamic programming [23] for mapping data flow trees (DFTs) into assembly code. The basic idea in this approach is to represent the target machine instruction set in the form of a cost-attributed tree grammar, and parsing each given DFT with respect to that grammar. As a result, an optimum derivation for the given cost metric, and hence an optimal code selection, are obtained. The runtime complexity is only linear in the DFT size.

For the implementation, we used the OLIVE tool (an extension of IBURG [24] contained in the SPAM compiler [25]), that generates code selector C source code for a given instruction set represented by a tree grammar. Specifying the instruction set with OLIVE is convenient, since the tool permits to attach *action functions* to the instruction patterns, which facilitates book-keeping and assembly code emission.

The LANCE frontend splits each C function into a set of basic blocks, each of which contains an ordered list of DFTs. The DFTs, which are directly generated in the format required for OLIVE, are passed to the generated code selector and are translated into assembly code sequences one after another. During this phase, also the calls to compiler-known functions (CKFs) are detected and are directly transformed into the corresponding NP assembly instructions. This step is rather straightforward, since CKFs are simply identified by their name. However, the code selector, in cooperation with the register allocator, is still responsible for a correct register mapping, since CKFs are called with symbolic C variables instead of register names. The result of the code selection phase is symbolic assembly code with references to virtual registers. This code is passed to the register allocator described in the following.

B. Register allocation

Although the NP shows a RISC-like basic architecture, the classical graph coloring approach to global register allocation [26] cannot be directly used. The reason is the need to handle *register arrays*. As explained in section III (see also Figs. 6 and 8), register arrays arise from indirect addressing in C programs, where unaligned bit packets are traversed within loops. As a consequence, virtual registers containing (fragments of) bit packet arrays have to be assigned to contiguous windows in the physical register file.

In order to achieve this, the register allocator maintains two sets of virtual registers: one for scalar values and one for register arrays. All virtual registers are indexed by a unique number, where each register array gets a dedicated, unique, and contiguous index range. As usual, register allocation starts with a lifetime analysis of virtual registers. Potential conflicts in the form of overlapping life ranges are represented in an *interference graph*, where each node represents a virtual register, and each edge denotes a lifetime overlap. The lifetime analysis is based on a standard def-use analysis of virtual registers [27].

During lifetime analysis, special attention has to be paid to bit packets indirectly addressed via register pointers, whose values might not be known at compile time. In order to ensure program correctness, all register array elements potentially pointed to by some register pointer p are assumed to be live while p is in use. Liveness of p is determined by inspecting the pointer initializations in the calls to compiler-known function *INIT* (see Fig. 8).

Due to the allocation constraints imposed by register arrays, the mapping of virtual registers to physical registers is based on a special multi-level graph coloring algorithm. Physical registers are assigned to those virtual registers first that belong to register arrays. This is necessary, since register arrays present higher pressure for the register allocator than scalar registers.

First, any node set in the original interference graph that belongs to a certain register array is merged into a *supernode*. Then, the interference graph is transformed into a *super-interference graph* (SIG), while deleting all edges internal to each supernode and all scalar virtual register nodes and their incident edges (Fig. 9).

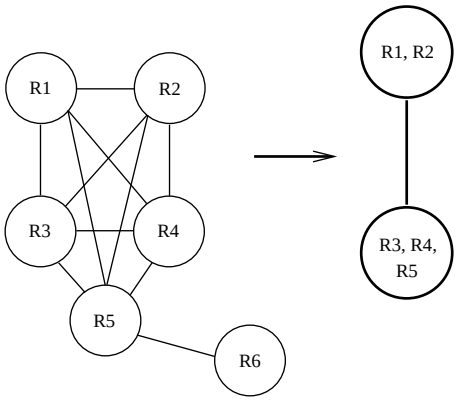

Fig. 9. *Construction of the SIG: In this example, the virtual register sets* $\{R1, R2\}$ *and* $\{R3, R4, R5\}$ *are supposed to represent two register arrays, while* $R6$ *refers to a scalar variable.*

Next, a weight is assigned to each supernode n, which is equal to the number of internal virtual registers of n plus the maximum number of internal virtual registers of n's neighbors in the SIG. The supernodes are mapped to physical registers according to descending weights. This heuristic is motivated by the fact that supernodes of a lower weight are generally easier to allocate, because they cause less lifetime conflicts. Furthermore, in case of a conflict, it is cheaper to spill/reload a smaller array instead of a larger one. For any supernode n with r internal virtual registers, a contiguous range in the register file is assigned. Since there may be multiple such windows available at a certain point of time, the selection of this range is based

on the best fit strategy in order to ensure a tight packing of register arrays in the register file, i.e. in order to avoid too many spills.

In our approach any element of a register array can be accessed in two different ways: first by direct addressing (e.g. `A[3]`) or indirectly by the use of a bit packet pointer. In case of insufficient physical registers using indirect access, spill code is generated for all virtual registers within a register array. Otherwise only the particular virtual register is spilled. After register allocation all symbolic addresses for bit packets have to be recalculated because now they specify a physical register within the register file instead of the name of a virtual register.

The permissible size of bit packet arrays is constrained by the size of the physical register file. Therefore no register allocation can be done for register arrays that are larger than the register file, and the compiler needs to reject such code with corresponding error messages. Note that for such code it would be at least very difficult to find a valid register allocation even manually. For certain source programs, one solution to this problem is to split large register arrays into smaller pieces already in the C source code and to perform computations on a set of small arrays instead of a single large one. However, this only works for programs, where the sub-arrays required at each program point are already known at compile time. Eventually, a solution can always be found by mapping bit packet arrays to memory just like regular data arrays, in which case the advantages of packet-level addressing are naturally lost in exchange for a safe fallback position.

After register allocation for the supernodes has been performed, all remaining virtual registers in the original interference graph are mapped to physical registers by traditional graph coloring [26], while inserting spill code whenever required.

V. Results

The C compiler for the NP described in the previous sections is fully operational. The performance of the generated code has been measured by means of a cycle-true NP instruction set simulator for a set of test programs.

As may be expected, the quality of compiler-generated code as compared to hand-written assembly code largely depends on the clever use of the CKFs and the underlying register array concept. When using CKFs without specific knowledge of the application, the performance overhead of compiled code may be several hundred percent, which is clearly not acceptable for the intended application domain. This is due to a massive increase in register pressure, when too many register arrays are simultaneously live, which naturally implies a huge amount of spill code.

On the other hand, a careful use of CKFs, as derived from detailed application knowledge, generally leads to a small performance overhead in the order of only 10 %. We observed that this overhead can be even reduced further by means of instruction scheduling techniques to reduce register lifetimes (and thereby spill code), as well as by peephole optimizations, which so far have not been implemented.

It is also interesting to consider the improvement offered by CKFs as compared to regular C code. Table I shows the performance for six small test routines. These mainly include packet oriented arithmetic operations on bit streams, as well as masking, counting, and checksum computations. The C programs have been compiled into NP machine code. Columns 2 and 3 give the simulated performance (clock cycles) of the compiled code without and with the use of CKFs and register arrays, respectively. Column 4 shows the performance gain in percent.

	without CKFs	with CKFs	gain %
prg1	33	24	27
prg2	41	29	29
prg3	43	29	33
prg4	853	639	25
prg5	103	81	21
prg6	156	106	32

TABLE I
Experimental performance results

The use of packet-level addressing resulted in an average performance gain of 28 % over the original C reference implementations without CKFs and register arrays. Naturally, this mainly has to be attributed to the NP hardware itself. However, from a system-level perspective it has been very important to prove that this performance gain can also be achieved by means of compiled C programs instead of hand-written assembly code.

As a result of our evaluation, we believe that the introduction of CKFs and register arrays represents a reasonable compromise between programming effort and code quality. CKFs give the programmer direct access to dedicated instructions which are important for optimizing the "hot spots" in a C application program, while the compiler still performs the otherwise time-consuming task of register allocation. For non-critical program parts, where high performance bit-level operations are hardly an issue, the productivity gain offered by the compiler versus assembly programming clearly compensates the potential loss in code quality.

VI. Conclusions and future work

We have outlined compiler challenges encountered for Network Processors, a new class of ASIPs, that allow for efficient protocol processing by means of packet-level addressing. We have described the implementation of a C compiler for a real-life industrial NP. The main concepts in this compiler, in order to make packet-level addressing accessible at the C language level, are the use of compiler-known functions and a special register allocation technique. Experimental results indicate that these techniques work in practice, so that the processor features are well exploited. Although the detailed implementation is machine-specific, we believe that the main techniques can be easily ported to similar NPs, for which a growing compiler demand may be expected in the future. An example is the Intel i960,

whose instruction set also partially supports bit packet addressing.

Improved versions of the NP C compiler are already planned. Ongoing work deals with gradually replacing the pragmatic approach of compiler-known functions with more sophisticated code selection techniques, capable of directly mapping complex bit masking operations into single machine instructions. This will be enabled by the use of special tree grammars that model the instruction set for the code selector. In addition, we plan to include a technique similar to register pipelining [29] in order to reduce the register-memory traffic for multi-register bit packets, and several peephole optimizations are being developed in order to further close the quality gap between compiled code and hand-written assembly code.

References

[1] Tensilica Inc., *http://www.tensilica.com*.

[2] Austria Mikro Systeme International, *http://asic.amsint.com/databooks/digital/gepard.html*, 2000.

[3] B. Wess, "Automatic Instruction Code Generation based on Trellis Diagrams," *IEEE Int. Symp. on Circuits and Systems (ISCAS)*, 1992.

[4] G. Araujo, S. Malik, "Optimal Code Generation for Embedded Memory Non-Homogeneous Register Architectures," *8th Int. Symp. on System Synthesis (ISSS)*, 1995.

[5] S. Liao, S. Devadas, K. Keutzer, S. Tjiang, A. Wang, "Code Optimization Techniques for Embedded DSP Microprocessors," *32nd Design Automation Conference (DAC)*, 1995.

[6] A. Timmer, M. Strik, J. van Meerbergen, J. Jess, "Conflict Modeling and Instruction Scheduling in Code Generation for In-House DSP Cores," *32nd Design Automation Conference (DAC)*, 1995.

[7] S. Bashford, R. Leupers, "Constraint Driven Code Selection for Fixed-Point DSPs," *36th Design Automation Conference (DAC)*, 1999.

[8] D.H. Bartley, "Optimizing Stack Frame Accesses for Processors with Restricted Addressing Modes," *Software – Practice and Experience, vol. 22(2)*, 1992.

[9] S. Liao, S. Devadas, K. Keutzer, S. Tjiang, A. Wang, "Storage Assignment to Decrease Code Size," *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 1995.

[10] R. Leupers, F. David, "A Uniform Optimization Technique for Offset Assignment Problems," *11th Int. System Synthesis Symposium (ISSS)*, 1998.

[11] E. Eckstein, A. Krall, "Minimizing Cost of Local Variables Access for DSP Processors," *ACM Workshop on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, 1999.

[12] R.J. Fisher, H.G. Dietz, "Compiling for SIMD Within a Register," *11th Annual Workshop on Languages and Compilers for Parallel Computing (LCPC98)*, 1998.

[13] R. Leupers, "Code Selection for Media Processors with SIMD Instructions," *Design Automation & Test in Europe (DATE)*, 2000.

[14] S. Larsen, S. Amarasinghe, "Exploiting Superword Level Parallelism with Multimedia Instruction Sets," *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2000.

[15] P. Paulin, "Network Processors: A Perspective on Market Requirements, Processors Architectures, and Embedded S/W Tools," *Design Automation & Test in Europe (DATE)*, 2001.

[16] D. Brooks, M. Martonosi, "Dynamically Exploiting Narrow Width Operands to Improve Processor Power and Performance," *High-Performance Computer Architecture (HPCA-5)*, Jan 1999.

[17] M. Stephenson, J. Babb, S. Amarasinghe, "Bitwidth Analysis with Application to Silicon Compilation," *ACM SIGPLAN Conference on Program Language Design and Implementation (PLDI)*, June 2000.

[18] S.C. Goldstein, H. Schmidt, M. Moe, M. Budiu, S. Cadambi, R.R. Taylor, R. Laufer, "PipeRench: A Coprocessor for Streaming Multimedia Acceleration," *26th Annual International Symposium on Computer Architecture (ISCA)*, 1999.

[19] S.C. Goldstein, H. Schmit, M. Budiu, S. Cadambi, M. Moe, R.R. Taylor, "PipeRench: A Reconfigurable Architecture and Compiler," *IEEE Computer*, vol. 33, no. 4, 2000.

[20] X. Nie, L. Gazsi, F. Engel, G. Fettweis, "A New Network Processor Architecture for High-Speed Communications," *IEEE Workshop on Signal Processing Systems (SiPS)*, 1999.

[21] C. Liem, "Retargetable Compilers for Embedded Core Processors," *Kluwer Academic Publishers*, 1997.

[22] A. Inoue, H. Tomiyama, H. Okuma, H. Kanbara, and H. Yasuura, "Language and Compiler for Optimizing Datapath Widths of Embedded Systems," *IEICE Trans. Fundamentals*, vol. E81–A, no. 12, pp. 2595–2604, Dec. 1998.

[23] A.V. Aho, M. Ganapathi, S.W.K Tjiang, "Code Generation Using Tree Matching and Dynamic Programming," *ACM Trans. on Programming Languages and Systems*, vol. 11, No. 4, 1989.

[24] C.W. Fraser, D.R. Hanson, T.A. Proebsting, "Engineering a Simple, Efficient Code Generator Generator," *ACM Letters on Programming Languages and Systems*, vol. 1, no. 3, 1992.

[25] A. Sudarsanam, "Code Optimization Libraries for Retargetable Compilation for Embedded Digital Signal Processors," Ph.D. thesis, Princeton University, Department of Electrical Engineering, 1998.

[26] P. Briggs, "Register Allocation via Graph Coloring," Ph.D. thesis, Dept. of Computer Science, Rice University, Houston/Texas, 1992.

[27] A.V. Aho, R. Sethi, J.D. Ullman, "Compilers - Principles, Techniques, and Tools," Addison-Wesley, 1986.

[28] R. Leupers, "Code Optimization Techniques for Embedded Processors," *Kluwer Academic Publishers*, 2000. Software: *http://LS12-www.cs.uni-dortmund.de/∼leupers*.

[29] D. Callahan, S. Carr, K. Kennedy, "Improving Register Allocation for Subscripted Variables," *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 1990.