Code optimizations for a VLIW-style network processing unit

Jinhwan Kim¹, Yunheung Paek²,*,† and Gangryung Uh³

¹Department of Electrical Engineering, KAIST, Daejon 350-701, South Korea
²School of Electrical Engineering, Seoul National University, Seoul 151-744, South Korea
³Department of Computer Science, Boise State University, Boise, ID 83725, U.S.A.

SUMMARY

The explosive growth in network bandwidth and Internet services such as QoS (quality of service) and SLA (service level agreement) monitoring have created the need for new networking hardware called a Network Processing Unit (NPU). In order to rapidly reconfigure the NPU for frequently varying Internet services and technologies, a high-performance C compiler is urgently needed. Several code generation techniques, which are intended to meet the high code quality demands of other types of application specific instruction-set processors (ASIPs) like digital signal processors (DSPs), have already been developed. However, these techniques are insufficient for NPUs due to striking architectural differences such as asymmetric data paths. The main purpose of this paper is to discuss our recent experience with the development of a commercial compiler for a new NPU called the Paion PPII, which is basically a packet engine for NPU to meet the growing need for new high-bandwidth communication equipment targeted for Internet routers and ethernet adapters. For this purpose, we will first show the architectural challenges posed by the target NPU. Then, we will describe several compiler techniques that we found to be effective for the target NPU with various unorthogonal architectural features. The current implementations of the PPII use a VLIW (Very Long Instruction Word) architecture. So, we handled this VLIW-style architecture by employing a simple code compaction scheme which packs multiple parallel instructions into one long instruction word. The experimental results show that our techniques are effective for significantly reducing the dynamic instruction count. Copyright © 2004 John Wiley & Sons, Ltd.

KEY WORDS: compiler; embedded system software; network processors; VLIW

*Correspondence to: Yunheung Paek, Department of Electrical Engineering, Seoul National University, Seoul 151-744, South Korea.
†E-mail: ypae@ece.snu.ac.kr

Contract/grant sponsor: KRF; contract/grant number: D00263
Contract/grant sponsor: ETRI
Contract/grant sponsor: KOSEF; contract/grant number: R08-2003-000-10781-0
Contract/grant sponsor: SNU
Contract/grant sponsor: NASA Idaho EPSCoR
Contract/grant sponsor: University IT Research Center Project
Contract/grant sponsor: Samsung Advanced Institute of Technology

Published online 20 April 2004
Copyright © 2004 John Wiley & Sons, Ltd.
1. INTRODUCTION

The need for higher network bandwidth and additional various Internet services has been increasing. This has created the need for new networking hardware to support not only higher network bandwidth but also programmability. Although traditional ASIC (Application Specific Integrated Circuit) based switching technology can support higher network bandwidth, this approach severely lacks programmability, which has led to the creation of new programmable ASIPs for network applications called Network Processing Units (NPUs). To meet the fast growing need of NPUs, many companies such as Intel, MMC, IBM and Agere Systems have introduced various NPUs since 2000 [1–6].

Programmability is an important requirement for NPUs along with performance. The NPU should be easily programmed to customize various feature sets. In addition, the NPU should be rapidly reconfigured to support frequently varying Internet services and technologies. For example, a networking hardware vendor, who is planning to provide a product that can classify traffic flows based on Layer 4 information, will demand a highly programmable NPU to rapidly implement such a feature. In order to support frequently varying services that service providers may offer, the NPU should be highly reprogrammable. Therefore, in order to meet such programmability requirements, NPUs commonly support HLL (High-Level Language) compilers.

A HLL compiler for a NPU should play the following two major roles to improve the competitiveness of embedded products containing the NPU. First, the compiler should relieve application programmers from the burden of using both time-consuming and error-prone assembly languages. Thus, fast time-to-market and reliability requirements for NPUs can be easily met. Second, the compiler should generate high-quality code to satisfy tight real-time performance constraints.

However, the specialized architectural features of NPUs frequently prevent the compiler from applying conventional optimizations to fully exploit the processor capabilities, including an unbalanced memory hierarchy, bit-stream operations and multi-processors. Several code generation techniques, which are intended to meet the high code quality demands of other types of ASIPs like DSPs (Digital Signal Processors), have already been developed. However, these techniques are insufficient for NPUs due to architectural differences between NPUs and other types of ASIPs. Therefore, it is vital to develop the code generation techniques that can exploit the NPU capabilities.

This paper reports in more detail our recent work [7] on compiler optimization for an industrial network processing unit based on the Zephyr compiler infrastructure [8]. Retargetability was one of the primary design goals of the Zephyr project. While Zephyr has been retargeted to numerous General Purpose Processors (GPPs) and recently retargeted to a DSP [9], its retargetability to NPUs has not been explored. Besides, there has been little work on retargeting existing compilers to NPUs. This is mainly because NPUs are relatively new and commercial products containing NPUs have not been available until recently [6,10].

In this work, we retargeted Zephyr to a commercial NPU by modifying its middleware, which is called a code expander. We also added to its code optimizer, called the Very Portable Optimizer (VPO), various optimization techniques specifically designed to improve the performance on our target

---

‡Satisfying the real-time performance means that a NPU can process a packet in a timely manner.

---

Copyright © 2004 John Wiley & Sons, Ltd. Software Practice and Experience 2004; 34:847–874
network processor architecture. Throughout this work, we analyze the problems that occurred in retargeting Zephyr to a NPU, propose our solutions and present our experimental results.

This paper is organized as follows. Section 2 outlines our target NPU architecture. Section 3 outlines the code generation and optimization process in the Zephyr compiler infrastructure. Section 4 shows how we modified Zephyr to generate code for the target architecture. Section 5 presents our recent experimental results. Section 6 describes related work in code generation for special NPU features. Section 7 concludes with a description of our ongoing research for this work.

2. THE PAION PPII PROCESSOR

Packet processing, towards which NPUs are specially tailored, has several salient characteristics compared with conventional data processing. One striking characteristic of packet processing is that a single data stream consists of a large number of individual packets. Each packet is independently processed through the following four basic tasks shown in Figure 1. First, during the parse stage, the contents of the packet header and fields are analyzed and classified. Second, during the search stage, tables are searched for a match between the content that has been classified and pre-defined content (or rules). Third, during the resolve stage, the destination and quality of service (QoS) requirements are resolved and the packet is routed to its destination. Finally, in the modify stage, whenever necessary, the packet is modified. Typically, a NPU is configured to have several packet processing optimized multi-processors called packet engines. Such a configuration results from the fact that independence between packets makes packet processing ideal for a multi-processor architecture. Each packet engine is responsible for accelerating the four basic packet processing tasks.

Additional characteristics of packet processing compared with those of data-processing, are lack of floating-point operations and lack of data locality among packet data. Thus, the configuration of a packet engine does not require a Floating Point Unit (FPU) and a data cache. Instead, the configuration includes complex bit-manipulation units to efficiently handle a bitstream-oriented packet protocol.

Many NPUs supplement packet engines with coprocessors, such as hash engines, search engines, classification engines and policy engines. They also include a GPP unit, called a control processor, as well as the packet engines. A widely known RISC processor is typically used as the control processor. In addition, NPUs have two major external interface types. One is the interface that connects

---

§NPU manufacturers use a variety of terms such as microengines, picoengines, channel processors, etc.
to a packet memory. The other is the interface that connects to a table memory, which is typically implemented as high-speed SRAM.

Many companies have introduced NPUs such as Intel’s IXP 1200 [2], MMC’s nP3400 [3] and IBM’s Rainier NPU [1]. In this work, we retarget Zephyr to Paion’s [4] PPII processor, which is a packet engine. Paion’s NPU follows a typical configuration of other NPUs such that it consists of multiple PPIIs along with SRAM shared among the packet engines.

As illustrated in Figures 2 and 4, the PPII shows a 32-bit embedded processor architecture with a central controller, host processor interface, instruction memory, instruction fetcher, decoder, data path controller, peripheral access controller, and 32-bit data path module.

The current implementations of the PPII use a Very Long Instruction Word (VLIW) architecture. This means that the PPII supports parallel processing to execute several atomic instructions within one cycle and can be optimized for memory access. A control instruction, a functional instruction and a memory instruction are packed into one 38-bit long instruction word, as shown in Figure 3(a). The SEQ and COND bits in Figure 3(a) indicate a control instruction, the AGEN and PACC bits indicate a memory instruction and the remaining bits indicate a functional instruction. This long instruction word is expressed in assembly as shown in Figure 3(b). The next() instruction is the control instruction which means that the next instruction word will be fetched. The agen() is the memory instruction which generates addresses and the swrite() is the memory instruction which stores to an external SRAM. The add() is the functional instruction that performs addition. PPII instruction words are separated by the control-related instructions encoded in SEQ bits such as next() and jcc().

Another feature of the PPII can be found in an ALU and register files, which comprise the data path module. The ALU is designed to execute complex bit operations as well as basic RISC-style operations. These complex bit operations in the PPII are the hfunc() instruction and align() instruction.
Figure 3. PPII VLIW instruction set architecture: (a) a long instruction word format; (b) assembly code example for a single long instruction word.

Figure 4. Simplified block diagram.
The `hfunc()` instruction is used to get a hash key by performing exclusive OR (XOR) operations on four operands. The `align()` instruction is used to pack a 48-bit packet of data into two 32-bit registers.

In the data path module of the PPII, the register file is heterogeneous such that registers are partitioned into general-purpose data registers for packet processing and transfer registers for memory access. Figure 4 shows the simplified data path module block diagram. This architecture has three buses—one is the destination bus for the ALU output and the others are left/right operand buses for the source operands of ALU. Using these buses, instructions can be executed simultaneously.

As shown in Figure 4, the data path to the external memory is through the operand bus so that the values stored in both sets of data registers and transfer registers can be moved to external memory. However, note that values stored in memory can be moved only to the transfer register set because there is no direct data path from memory to other registers except transfer registers. This configuration allows the processor to execute a functional instruction and a memory instruction in parallel. In addition, the chip size can be effectively reduced by eliminating otherwise required additional control units.

PPII has a complex memory architecture like many other NPUs such as Intel’s IXP 1200 [2]. Multiple PPIIs comprise a NPU with one SRAM in order to share the lookup table information. The PPII has an ingress buffer for packet data and an interface connected to SRAM for a lookup table as well as program memory in which firmware code is stored. SRAM is externally located to each PPII since it must be uniformly accessed by multiple PPIIs.

3. THE ZEPHYR INFRASTRUCTURE

The Zephyr compiler infrastructure consists of three major modules, which are the frontend, code expander, and VPO [8]. VPO is the backend that performs machine-dependent optimizations and code generation for a given target architecture.

The design goal for Zephyr, as a part of the National Compiler Infrastructure, is to provide compiler writers with a retargetable compiler platform which can be used to build code generators for many different processors at a low development cost. To achieve this retargetability goal, we have VPO as a core module in the center of the Zephyr infrastructure.

Unlike most other backend optimizers, the uniqueness of VPO lies in its interesting property that all its optimizations are machine independent, yet performed on a machine-level instruction form. This property allows compiler writers to reuse most of the transformations in VPO, while suffering relatively little performance loss when they build compilers for different targets. Zephyr’s retargetability is made possible by its pervasive use of a low-level intermediate representation (IR), called Register Transfer Lists (RTLs) [12].

RTLs are a tree-like well-defined IR that glues all the compiler modules of Zephyr in a uniform fashion. It is a simultaneous composition of a list of register transfer expressions, each of which corresponds to a single effect of the form

\[ lvalue = rvalue \]

where the `rvalue` expression is evaluated and stored to the `lvalue` location. An RTL can support operators with an arbitrary number of operands, depending on their types, and operands can be either symbolic registers, memory locations or constants.
Register transfer expressions are primitive enough to represent in composition almost all possible machine operations and addressing modes provided by various existing architectures. The following list shows example SPARC instructions and the corresponding RTLs:

1. \( \text{add} \ %o2,%i1,%o0 \ \ r[8]=r[10]+r[25]; \)
2. \( \text{sub} \ %o0,%o2,%l0 \ \ r[16]=r[8]−r[10]; \)
3. \( \text{mov} \ %o2,%o0 \ \ r[8]=r[10]; \)
4. \( \text{mov} \ %l0,%o1 \ \ r[9]=r[16]; \)
5. \( \text{call} \ t2 \ \ ST=t2. \)

Each register transfer expression represents the semantics of target machine instructions. By hand-writing a code expander, the front-end’s IR is translated to the RTL-based control-flow graph.

A code expander translates each intermediate operation into a sequence of RTLs. The key to manually preparing code expanders is to keep the output RTLs as simple as possible, where complex instructions and addressing modes are broken down into primitive RTL operations (e.g. + or −) and modes (e.g. register or register indirect addressing), respectively. For example, if we need to add a constant value, say 4, and a value from a variable stored at an offset, say 8, from the stack pointer, we can just produce the following RTL in the code expander, where \( t_i \) stands for a temporary value stored in a pseudo register.

1. \( t_1 = 4; \)
2. \( t_2 = 8; \)
3. \( t_3 = SP; \)
4. \( t_4 = t_2 + t_3; \)
5. \( t_5 = M[t_4]; \)
6. \( t_6 = t_1 + t_5. \)

By keeping a code expander simple, the retargeting of a compiler to a new architecture can be simplified in Zephyr. The RTL output code generated by a code expander is usually naive since it may not yet capitalize on some important target instructions. However, this initial code can be transformed to more efficient final target machine code by machine-independent transformations in VPO. For instance, from the naive sequence of RTLs shown above, VPO can easily discover a more efficient form: \( t_6 = 4 + M[SP + 8]. \)

Figure 5 shows the optimization sequence in VPO. As stated earlier, to be optimizable by VPO, a program must be expressed as a set of procedures, and each procedure must be expressed as a control-flow graph in which the individual elements are Zephyr RTLs. Such a program has well-defined semantics, which is drawn from the RTL semantics. Given any such program, VPO repeatedly improves it until no additional improvements are found.

In VPO, even an optimization written to exploit a certain architectural feature of a single machine can be expressed in the machine-independent framework. If a new machine incorporates the same feature, VPO automatically exploits it. Many machine-dependent optimizations are already represented in machine-independent manner as transformations of flow graphs containing RTLs, so that the retargeting efforts to other platforms can be dramatically reduced. For example, consider the post-increment operation \( i++ \) from a C statement \( a[i++] = 1 \). The code expander will translate this statement into the naive RTLs shown in Figure 6(a). Then, VPO replaces memory accesses with free registers as shown in Figure 6(b). Afterwards, VPO repeatedly follows UD-chains to combine a set of
Optimize(program) {
    · · · Control and data analysis · · ·
    FindLoops()
    EstimateExecutionFrequency()
    repeat
        repeat
            LiveVariableAnalysisUpdate()
            DeadVariableElimination()
            if ColorLocalVariables() then
                InstructionSelection()
            endif
        until reached fixed point or used all registers
        if changed then
            CommonSubexpressionElimination()
            LiveVariableAnalysisUpdate()
            DeadVariableElimination()
            LoopTransformations()
            InstructionSelection()
            InlineFunctions()
        endif
    until reached fixed point or used all registers
    ControlFlowTransformations()
    InsertFunctionPrologueandEpilogue()
    InstructionSelection()
    InstructionScheduling()
}

Figure 5. Code generation/optimization routines in VPO.

\begin{verbatim}
   r[0]=R[SP+i];    \hspace{.5cm} r[0]=r[1];    
   R[SP+t0]=r[0];  \hspace{.5cm} r[2]=r[0];  r[0]    
   r[0]=R[SP+i];    \hspace{.5cm} r[0]=r[1];    
   r[0]=r[0]+1;     \hspace{.5cm} r[0]=r[0]+1
   R[SP+i]=r[0];   \hspace{.5cm} r[1]=r[0];  r[0]
   r[0]=R[SP+t0];  \hspace{.5cm} r[0]=r[2];  r[2]
   R[r[0]]=1;      \hspace{.5cm} R[r[0]]=1;  r[0]

   (a) \hspace{2cm} (b)

   r[2]=r[1];  \hspace{.5cm} r[2]=r[1];
   r[1]=r[1]+1; \hspace{.5cm} R[r[1]]=1; r[1]=r[1]+1;
   R[r[2]]=1;  \hspace{.5cm} r[2]

   (c) \hspace{2cm} (d)
\end{verbatim}

Figure 6. Code combining in VPO: (a) initial RTLs; (b) after register allocation;
(c) after instruction selection; (d) after more instruction selection.
related RTLs into a new one and tests if the new RTL forms a valid machine instruction. If the test fails, the transformations combining instructions are rolled back. Figure 6(c) shows the resulting RTLs after combining instructions. Many machines support auto-increment/decrement addressing modes and if the target machine also supports the addressing modes, then the RTLs in Figure 6(d) will again be combined to form a single RTL shown in Figure 6(d), which will be later translated to an auto-increment instruction in the target.

4. OPTIMIZATION TECHNIQUES FOR THE TARGET NETWORK PROCESSING UNIT

The Zephyr compiler infrastructure has been retargeted across a wide spectrum of GPPs. It has also been retargeted to some DSPs, such as TI TMS320C5402 DSP [9], by adding DSP-specific code optimization techniques. However, the compiler has never been retargeted for any NPUs that typically exhibit unique architectural features for packet processing such as bit operations.

In this section, we will explain our first Zephyr compiler retargeting effort to the network architecture of the PPII. First, we briefly describe several important hardware features that distinguish the PPII from both GPPs and other types of ASIPs like DSPs with respect to code generation. Second, we explain salient limitations of the original implementation of Zephyr to exploit such architectural features. Finally, our compiler optimization techniques are presented to circumvent these limitations.

4.1. Heterogeneous register architecture

Many compilers, including GCC and Zephyr, perform instruction selection and register allocation in different optimization phases as shown in Figure 7(a). That is, during the instruction selection phase, instructions reference pseudo registers which, during the subsequent register allocation phase, are allocated to any available physical registers. Decoupled optimizations perform well for GPPs that have relatively homogeneous registers with the exception of a small set of special registers.

Albeit this strategy is conceptually simple, it does not work well for ASIPs since the architectures have heterogeneous registers and the permissible registers for the instructions are even further restricted due to limited instruction encoding length. That is, as shown in Table 1, when an instruction is selected, the storage types of an operand should be selected or registers should be allocated. Therefore, for ASIPs, register allocation and instruction selection should be performed in a tightly coupled manner, as shown in Figure 7(b).

In Zephyr, when a code expander generates specific machine-level RTL instructions, register allocation is postponed by simply assigning pseudo registers to store the temporary values. These pseudo registers are later allocated to physical ones. As stated earlier, these decoupled phases of instruction selection and register allocation processes make Zephyr, let alone code optimizations, inappropriate to generate efficient code for processors which contain heterogeneous registers.

One general solution to overcome the limitations of Zephyr would be to rewrite a whole new code generation algorithm that combines instruction selection and register allocation in one phase, as is done

---

¶A stack pointer and a program counter can be separately handled as special cases.
in many other compilers for DSPs [13]. However, this would require too much retargeting effort of any compiler designers who want to retarget their compilers like Zephyr.

As an alternative approach, we modified Zephyr by adding a new scheme which enforces an extra relationship that combines the two phases in the following manner. During the instruction selection phase, permissible types of registers are also identified for each instruction. The subsequent register allocation attempts to assign physical registers with in each identified type of registers. In other words, when generating RTL instructions, a code expander is modified to assign the type of registers suitable for each RTL instruction in order to use the register allocation phase in VPO without any modification. Table II shows the RTLs generated by code expander with this method.

To show how a code expander generates RTLs, consider the RTLs that are generated from C code as shown in Table II. Notice that the variable a in the first C statement takes a constant value. In order
to translate this statement, a code expander selects an instruction that loads an immediate value to a
register as shown in row 1 of Table II, and a store instruction that stores the value in the register
to memory in row 2 of the same table. The code expander then assigns a data register type to the
destination of the immediate load instruction and the source of the store instruction. In order to translate
row 2 of Table II, load instructions that load the variable a from memory are necessary. Hence, the
code expander selects load instructions and assigns a transfer register type, MBR, as the corresponding
destination. The reason for assigning a transfer register type is due to the restricted data path from the
memory, as shown in Figure 4.

4.2. Asymmetric data paths

Similarly to the GNU GCC compiler, one salient feature of VPO is that it can invoke the instruction
selection and register allocation phases iteratively during the optimization process, as discussed in
Section 3. One advantage of this iterative methodology is that it makes the compilation process
more flexible and easier to apply additional optimization techniques. The code quality in Zephyr,
thus, can be gradually improved until certain desired requirements are met. By controlling these
requirements and changing appropriate modules in VPO, we can adjust the code quality in a flexible
way. Another advantage is that alternatively repeating invocation of the separate code optimization
phases helps VPO to compensate to some degree for the inefficiency due to the decoupled structure of
its phases.

However, to take advantage of this iterative compilation process, VPO requires the RTL code to
satisfy a certain property, called a machine invariant. To satisfy this property, every RTL in the code
should always match one instruction defined in the target machine. This property is useful to the
iterative compilation process because the process can be terminated at any time after trivially emitting
machine instructions for their matching RTLS as soon as certain requirements on code quality are met.
However, we found that this property can be a problem for compilers targeting NPUs which usually
have asymmetric data paths. Figure 4 shows a typical example of an asymmetric data path where there
is no data path from memory to data registers, while a data path from data registers to memory exists.
Conversely, GPPs for which VPO was originally designed usually have symmetric data paths; that is,
data paths between every register and memory are bi-directional so that any data stored to memory
from some register can always be directly loaded back to the same register.

When C code is translated to IR form, many compilers initially have many redundant loads and
stores, most of which can be later eliminated by optimization phases. For instance, Table III shows the
RTLS for the SUN SPARC that a code expander initially generates from the two C statements. Note in
the example that the RTLS have the machine invariant property because each RTL matches a single
SPARC instruction. Since the SPARC has symmetric data paths, the RTLS use the same registers to
move data from/to memory. Therefore, RTLS, each of which respectively loads or stores a value for the
same register, can be eliminated or combined into a fewer number of RTLS, as illustrated in Figure 8(a).

Table IV shows the RTL code generated for the PPII which also initially satisfies the machine
invariant property. Although VPO tries to eliminate redundant memory accesses for this code, it fails
because there are no RTLS which can be combined into a new RTL that satisfies the machine invariant
property by matching a PPII machine instruction. For instance, VPO would first attempt to combine
three RTLS M[a]=r[0];, mbr[0]=M[a]; and r[0]=mbr[0]; into one RTL r[0]==r[0];
so that the redundant memory access to a can be eliminated. However, it fails without additional
Table III. C, RTL and machine code for SPARC.

<table>
<thead>
<tr>
<th>C code</th>
<th>RTLs</th>
<th>Pseudo machine instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a = 0xc0;)</td>
<td>(r[0] = 0xc0;)</td>
<td>(r0 \leftarrow 0xc0)</td>
</tr>
<tr>
<td>(M[SP+a] = r[0];)</td>
<td>(M[SP+a] \leftarrow r0)</td>
<td></td>
</tr>
<tr>
<td>(r[0] = M[SP+a];)</td>
<td>(r0 \leftarrow M[SP+a])</td>
<td></td>
</tr>
<tr>
<td>(t = a &amp; \cdots;)</td>
<td>(r[1] = r[0] &amp; \cdots;)</td>
<td>(r1 \leftarrow r0 &amp; \cdots;)</td>
</tr>
<tr>
<td>(M[SP+t] = r[1];)</td>
<td>(M[SP+t] \leftarrow r1)</td>
<td></td>
</tr>
</tbody>
</table>

Figure 8. Code combining for SPARC and PPII: (a) SPARC; (b) PPII.

techniques since there is no valid PPII instructions that move data via direct data paths from the data register \(r[0]\) to the transfer register \(mbr[0]\) or from memory to \(r[0]\). This desired effect might be achieved during the CSE technique that chooses the cheapest expression for the fourth RTL \(r[0] = mbr[0];\). Instead, we handled these asymmetric data paths by describing additional data paths, called virtual data paths, to our target NPU architecture, so that the whole data paths look symmetric to the compiler. The virtual data paths help VPO to further combine the RTLs. For instance, a virtual data path from memory to data registers in the PPII is added to the original PPII data path shown in Figure 4. This new data path leads to the final RTLs which do not contain any redundant memory loads and stores as shown in Figure 9. Therefore, with virtual data paths, the VPO combiner can merge additional RTLs which are not possible in the presence of asymmetric data paths. Thus, the global CSE is potentially benefitted from these additional RTL merges.

Even after code combining, some RTLs using the virtual data paths may still remain in the compiler-generated code. Since these RTLs do not use real paths, they should be eliminated from the final output code. In this work, after the optimization process, VPO splits an RTL that directly loads to data...
Table IV. RTL and Machine code for the PPII.

<table>
<thead>
<tr>
<th>C code</th>
<th>RTLs</th>
<th>Pseudo machine instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a = 0xc0;) (r[0]=0xc0;) (M[a]=r[0];)</td>
<td>(r0 \leftarrow 0xc0)</td>
<td>M[a] (\leftarrow r0)</td>
</tr>
<tr>
<td>(mbr[0]=M[a];) (x[0]=mbr[0];)</td>
<td>(mbr0 \leftarrow M[a])</td>
<td>(r0 \leftarrow mbr0)</td>
</tr>
<tr>
<td>(t = a &amp; \cdots;) (r[1]=x[0] &amp; \cdots;)</td>
<td></td>
<td>(r1 \leftarrow r0 &amp; \cdots)</td>
</tr>
<tr>
<td>(M[t]=r[1];)</td>
<td>(M[t] \leftarrow r1)</td>
<td></td>
</tr>
</tbody>
</table>

Figure 9. Code combining for the PPII with virtual data paths.

Figure 10. Code generation in VPO with virtual data paths.

registers into two RTLs: the first RTL is to load into a transfer register and the second is to move a value from a transfer register to a data register. Simultaneously with splitting, an available transfer register is allocated. Figure 10 illustrates the overall process of combining instructions using virtual data paths.

### 4.3. Bit packet processing

Many ASIPs, such as DSPs and NPUs, offer complex instructions which are uncommon in RISC-style GPPs in order to accelerate the specific complex operations. However, C operations are restricted to
void L3PktModify(ULONG best_bkt) {

    tmp1 |= (tmp2 & 0x30) << 2;
    tmp1 |= (tmp2 & 0x0e) >> 1;
    tmp1 |= 0x20;

    tmp1 = (tmp2 & 0x01) << 31;

    tmp1 = (tmp2 >> 6) & 0x03FF00;
}

Figure 11. Bit manipulation example from L3 firmware code.

primitive operations such as addition (+), subtraction (−), and shift (<<, >>). The gap between high-level language (HLL) instructions and assembly instructions has become wider as ASIP architectures have supported more complex instructions. This makes code generation more difficult. The main reason is that a complex instruction cannot be expressed in the C language. Figure 11 presents an example code where complex bit operations for packet processing are expressed in the C language. Due to this complexity, the development of NPU’s applications is error prone and the code readability is poor so the code is not reusable. In addition, because of C programming style, repeated bit operations are made to a function. This results in frequent function calls so that the saves and restores of registers can too often occur for a calling convention. Thus, the compiler-generated code may not satisfy the required real-time performance.

Hence, the compiler should generate code by exploiting patterns for the complex instructions in RTLs. It is often relatively simple to exploit some patterns and emit the corresponding instructions; for example, in the case of a MAC instruction on DSPs, the compiler only finds the pattern which consists of * and + operations and any processing to operands is unnecessary. But the complex instructions on NPUs, which require patterns consisting of sophisticated bit operations to handle bitstream-oriented data for efficient packet processing, are usually too difficult to express in a HLL.

4.3.1. Bit instructions

Table V shows some of the complex bit-processing instructions in the PPII. You can see that each source operand of operations is not a match for a whole register as shown in Table V. For example, the hfunc() instruction takes two registers as sources, but each register is split into high and low portions as source operands. That is, the number of source operands that should appear in the C code is not two but four. This means that in order to find the pattern for hfunc() by using the same method used for a MAC instruction, the compiler needs to find three XOR ⊕ operations and the corresponding four C variables first of all. Even if this pattern is found, generating hfunc() depends on whether the values stored in variables can be encoded to 16-bit locations. If possible, the compiler generates not only hfunc() but also additional instructions for masking and shifting in order to split registers into two portions. Therefore, even when such a pattern-matching method is successfully applied to generate hfunc(), the code size would be still increased.

Copyright © 2004 John Wiley & Sons, Ltd.
Software Practice & Experience 2004; 34:847–874
Table V. Complex instruction examples.

<table>
<thead>
<tr>
<th>Assembly syntax</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>align(32,DIMM,LOP,ROP);</td>
<td>R0 = [LOP,ROP] &gt;&gt; DIMM[4:0]</td>
</tr>
</tbody>
</table>

In Table V, the `align()` instruction takes two registers as arguments. However, two registers are handled as a single operand in order to shift together. Hence, `align()` is also too complex to be expressed in the C code. Besides, the pattern for `align()` can be translated when the shift amount in DIMM is known at compile time because the value in DIMM must be encoded into an instruction word. Therefore, using such a pattern-matching method to generate `align()` is often very challenging.

4.3.2. Bit instructions using compiler intrinsic functions

As stated earlier, PPII offers complex instructions for complex bit operations. But these complex instructions are not easily generated from the C language. Thus, alternative methods such as using in-line assembly or supporting the complex instructions by means of libraries are needed for a compiler to efficiently generate the complex instructions. However, our goal is that a PPII compiler should generate the firmware code without libraries. In the case of in-line assembly, a programmer allocates hardware registers directly, so it cannot take advantage of using HLL programming. Therefore, in this work, intrinsic functions, also known as compiler-known functions (CKFs), were used. When using intrinsics, the programmer does not need to directly program the body of an intrinsic function. The compiler translates intrinsic calls into fixed instructions or instruction sequences [10, 14]. The intrinsic approach does not have any calling overhead. Thus, intrinsics offer the ability to efficiently utilize complex bit instructions. Readability of the C code is improved when using intrinsics, so the code can be more reusable. Figure 12 shows the format of intrinsics for complex bit instructions of Paion’s PPII.
hashkey = gen_hashkey(13, mac01, mac02);

```c
next();

hfunc(32b,r0,r8,r9);
// generate a hash key from r8, r9 and move its value into r0
next();

shift(32b,r3,r0,#3);
```

Figure 13. The intrinsic example for complex instructions.

The `gen_hashkey()` is an intrinsic function to get a 16-bit hashkey value from parameters, `data1` and `data2`. The compiler translates a `gen_hashkey()` into the `hfunc()` instruction and allocates parameter variables to registers. Figure 13 illustrates the resulting code generated from the intrinsic function. The intrinsic `align()` is translated into the `align()` assembly instruction and the compiler allocates the parameter variables, `data1` and `data2`, to registers. Therefore, as illustrated earlier, the intrinsic approach provides a convenient programming method as well as the ability for programmers to efficiently utilize complex bit instructions. Also, in the case of intrinsics, unlike inline assembly, the compiler performs register allocation, which leads the compiler to apply other optimizations [10].

4.4. Memory constraints

NPUs have specialized features different from conventional processor architectures. The main difference is that NPUs usually have multi-processor architectures as described in Section 2. In DSPs, the emphasis has been to use a complex-instruction solution to accelerate specific applications. However, NPUs are designed as multi-processor architectures to process several packets simultaneously. The multi-processor architecture makes the memory hierarchy system of NPUs differentiated from that of GPPs and DSPs. While a uni-processor architecture is able to have on-chip data SRAMs, a multi-processor architecture of NPUs is designed to share an external SRAM because, as mentioned earlier, all packet engines must share the information such as lookup table entries. Due to the use of external memory, memory optimizations that reduce the number of memory accesses are much more important to meet the required real-time performance than DSPs and GPPs, which usually have a fast on-chip SRAM or cache.

Also each PPII has another type of memory called an ingress buffer for incoming packet information. These types of memory in the PPII are called *peripherals*. However, a PPII has no memory types such as those found in a typical von Neumann architecture. So, an *activation record*, which includes local variables, can only reside in a part of SRAM by using the instructions in Table VI, which shows the PPII assembly instructions for peripheral accesses to these two types of memory. As described earlier, the SRAM is next to a PPII processor, so each memory access to SRAM takes about three cycles. Therefore, all memory accesses other than lookup table entries and packet data may degrade the overall performance substantially. As stated in Section 4.2, reducing the memory accesses for variables or temporaries was achieved by applying the virtual data paths to the compiler for the PPII. However, Zephyr does not support interprocedural register allocation, so function calls still...
## 4.4.1. Peripheral accesses using compiler intrinsic functions

Two types of memory in the PPII, an ingress buffer and a SRAM, cannot be separately expressed in the C language because the semantics of the C language assumes a von Neumann instead of a Harvard architecture, which has a program memory and one or more banks of data memory. Therefore, to distinguish these two types of memory, we applied methods using compiler intrinsic functions.

Figure 14 shows the format of intrinsics for peripheral access. The `data_ID` parameter selects the memory type, `byte_offset` indicates the memory address, and `MOS` indicates the operand size. The `get_data()` function loads the value stored in memory and the `set_data()` function stores the value stored in the `src_var` parameter into memory.

Using these intrinsics with the above stated advantages and assistance from the compiler, the memory can be used more efficiently because the `MOS` parameter of the intrinsic function enables a bit-level addressing. Figure 15 illustrates the assembly code generated from the intrinsic function for the peripheral access. The compiler translates the intrinsic functions written in C code into the immediate load instruction `imm()` and the address generation instruction `agen()` in order to calculate the memory address. The `PIF` parameter in Figure 15 indicates that the memory type is an ingress buffer. As shown in Figure 15(a), the `iread0()` instruction is emitted to move the value from memory to the `mbr0` transfer register, and then this value is moved to the `r3` data register by the `movl()` instruction.

### Table VI. Assembly instructions for peripheral access.

<table>
<thead>
<tr>
<th></th>
<th>Ingress buffer</th>
<th>SRAM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load</td>
<td><code>iread0(MOS);</code></td>
<td><code>sread0(MOS);</code></td>
</tr>
<tr>
<td></td>
<td><code>∼ iread3(MOS);</code></td>
<td><code>∼ sread3(MOS);</code></td>
</tr>
<tr>
<td>Store</td>
<td><code>iwrite(MOS,ROP);</code></td>
<td><code>swrite(MOS,ROP);</code></td>
</tr>
</tbody>
</table>

---

punkt_word2 = get_data(PIF, 8, 32);  set_data(PIF, 0, 32, pkt_word0);
next();  ↓
imm(r0, #8);
next();
agen(ea1 = aluout);
iread0(32b);
next();
movl(32b, r3, mbr0, aluout); // r[3] = mbr[0];
(1)

As shown in Figure 15(b), the iwrite() instruction is emitted to move the value from the r5 data register, to memory.

4.4.2. Interprocedural techniques

In C programming, functions are generally used for repeated operations. This enables us to develop the code in the C language more conveniently and enhances the readability and the reusability of the code. In the case of NPUs such as a PPII, the firmware code typically contains the repeated packet processing. Therefore, to facilitate convenient programming, the firmware code may be developed to contain many functions. One of the factors leading to wide memory use is frequent function calls. In ordinary C compilers, the activation record of a function is kept in memory and the information of the function such as the local variables is stored in the activation record. When a function is invoked, the caller needs to save the information of the caller into the activation record, and then when the function returns, the caller needs to restore its information of the caller back into the registers. In addition, the parameters and the return value are passed through memory. The compiler for conventional microprocessors, which have an on-chip cache or an on-chip SRAM, work well using this method because the memory latency is usually only one cycle. But this approach may cause serious performance degradation in the PPII which has no on-chip memory. To avoid frequent memory access, several techniques were considered. In our PPII compiler, the memory access instructions were reduced by means of the virtual data path algorithm, as mentioned in Section 4.2. Also the following techniques were applied.

(1) Passing parameters and return values via registers. Generally the argument area in the activation record of the callee is used for passing parameters. In the same way, the argument area in the activation record of the caller is used for passing back the return values. Since the activation records are allocated in memory, this argument passing process involves memory stores and the subsequent memory references to the incoming parameters in the callee. This means that it may fail to satisfy the required real-time performance of the PPII. To reduce the overall number of memory stores and the subsequent memory references for the argument passing process, we reserve five registers to pass up to four parameters and one return value. We call the four registers...
next();
  imm(r1,#5);  // aluout.imm and r0 = (immediate value)
next();
  agen(ea1=base+imm);  // address generation
  swrite(32b,rc);  // memory writing
next();
  movl(32b,rc,rb,aluout);  // r[c] = r[b]; aluout is meaningless.
next();
  cond(1);  // always (unconditional)
  call(Fpkt_flooding_srcFAIL);  // jump
next();
  imm(r1,#5);
next();
  agen(ea1=base+imm);
  sread3(32b);
next();
  movl(32b,rc,mbr3,aluout);  ...
Figure 17. Call graph with allocated registers: (a) before applying IRR; (b) after applying IRR.

We found that function inlining is useful to our compiler for PPII because the overhead of function calls can be drastically reduced for packet processing code. However, inlining generally also increases code size, which can be a critical drawback for NPUs with such a small internal program memory. So, in our work, only functions with a limited body size were inlined to avoid a code size explosion. The full version of the function inlining for VPO can be found in the literature [15].

(4) Interprocedural register renaming. For the cases where function inlining cannot be applied, we found that interprocedural register allocation [16] is useful for NPUs. Consider the example shown in Figure 17. Figure 17(a) gives the example of allocation results and a call graph. We can see in the example that we need 15 temporaries to be allocated to data registers. While each PPII has 16 data registers, the number of data registers allocated to these temporaries is only five, as can be seen in Figure 17(a). That is, while 11 data registers remain free, the same data registers are allocated in different functions. Hence, the data registers need to be repeatedly saved into the activation record when these functions are invoked. Figure 17(b) shows the optimized result of register allocation to overcome this problem. But the original implementation of Zephyr performs intra-procedural register allocation so registers could not be allocated as shown in Figure 17(b). In this work, Zephyr was extended to support the interprocedural register allocation by register renaming in order to get the result as shown in Figure 17(b). This allocation algorithm is performed on the final RTLs after all other optimizations have been applied, and simply changes the registers that overlap the registers in other functions along the path of the caller–callee chain into available registers. Figure 18 shows the detailed algorithm.

4.5. VLIW code generation

As mentioned in Section 2, a single instruction word consists of a control instruction, a functional instruction and a peripheral instruction. That is, a single instruction word does not consist of
\( R_f \) : register usage in function \( f \)  
\( S_f \) : register usage in ancestors and descendants of function \( f \)  
for each function \( f \)  
begin  
\( R_f = 0 \)  
\( S_f = 0 \)  
end  
for each function \( f \)  
for each register \( r \) used in \( f \)  
\( R_f = R_f \cup r \)  
do  
for each function \( f \)  
for each function \( g \) called by \( f \)  
\( S_g = S_g \cup R_f \)  
for each function \( g \) calls \( f \)  
\( S_g = S_g \cup R_f \)  
until no change  
for each function \( f \)  
if(\( S_f \cap R_f \neq \) empty)  
for each register \( r \) in \( S_f \cap R_f \)  
find candidate for \( r \)  
if candidate \( c \) exists  
replace all \( r \) in \( f \) with \( c \)  
\( R_f = (R_f - r) \cup c \)  
fi  
fi  
until \( R \) do not change

Figure 18. The algorithm for interprocedural register renaming.

the same instruction types. Fortunately, we found that in the final resulting RTLs, there are few sequences of control instructions or peripheral instructions because many peripheral instructions can be eliminated and control instructions are only expressed explicitly when jump or subroutine call occurs. The \texttt{next()} instruction means sequential execution so that it need not be expressed explicitly in RTLs. So, we extended Zephyr to simply pack a control instruction, a functional instruction and a peripheral instruction into a single long instruction word, as shown in Figure 3(b). Figure 19 shows the detailed algorithm.

For example, consider the final RTLs in Table VII. The first three RTLs are functional instructions executed sequentially, so a \texttt{next()} instruction is generated between each functional instruction as shown in Table VII. In the last RTL, a conditional instruction performs a jump to L13 when the result of the above RTL is not zero. Hence, \texttt{cond()} and \texttt{jcc()} instead of \texttt{next()} are generated as shown in Table VII.
for each function \( f \)
begin
\( \text{no}_f = 0 \)
\( \text{no}_p = 0 \)
\( \text{pRTL} = \text{the first RTL in } f \)
do
if (\( \text{pRTL} \subset \text{control instruction} \))
\( \text{no}_f = \text{no}_p = 0 \)
pack all unpacked RTLS including \( \text{pRTL} \)
else
if (\( \text{pRTL} \subset \text{functional instruction} \))
\( \text{no}_f = \text{no}_f + 1 \)
fi
if (\( \text{pRTL} \subset \text{peripheral instruction} \))
\( \text{no}_p = \text{no}_p + 1 \)
fi
fi
if (\( \text{no}_f > 1 \) | \( \text{no}_p > 1 \))
\( \text{no}_f = \text{no}_f - 1 \)
\( \text{no}_p = \text{no}_p - 1 \)
pack all unpacked RTLS excluding \( \text{pRTL} \)
print next() assembly instruction previous to \( \text{pRTL} \)
fi
\( \text{pRTL} = \text{the next RTL of } \text{pRTL} \)
until all RTLS are processed
end

Figure 19. The algorithm for packing into a single long instruction word.

Table VII. Example of code generation for VLIW.

<table>
<thead>
<tr>
<th>Final RTLS</th>
<th>Generated assembly code</th>
</tr>
</thead>
<tbody>
<tr>
<td>( r[5] = r[5] + 1; )</td>
<td>\text{add}(32b, r5, r5, #1); \text{next}();</td>
</tr>
<tr>
<td>( r[0] = 10; )</td>
<td>\text{imm}(r0, #10); \text{next}();</td>
</tr>
<tr>
<td>( \text{IC} = r[0] ? r[5]; )</td>
<td>\text{sub}(32b, \text{null}, r0, r5); \text{next}();</td>
</tr>
<tr>
<td>( \text{PC} = \text{IC} ! 0, \text{L13}; )</td>
<td>\text{cond}(ZF1); \text{jcc}(N, \text{L13});</td>
</tr>
</tbody>
</table>
5. EXPERIMENTAL RESULTS

We evaluated our PPII compiler with a suite of Paion’s firmware programs on a PPII simulator. We take C code as the source and, as the target, produce assembly code for the PPII which is in turn given to the assembler to produce the machine code. The performance of the generated machine code has been measured by means of the simulator, which was developed by Paion and displays code size and dynamic instruction count as the result.

In this section, we report our recent experimental results and analyze the effect of each major compiler technique on the performance.

5.1. Firmware programs for the PPII

To test the effectiveness of our techniques implemented for the PPII, we used a suite of firmware programs, which were internally developed by Paion for the PPII, as our benchmarks. Firmware should be stored in the limited program memory which shares SRAM with the lookup table. So the firmware code size is very important.

The Paion firmware, which is typically a network processing application, is classified into two parts: initialization programs and packet processing programs. Initialization programs run only once at first in order to initialize the PPII. Each of these programs performs some sequence of operations including frequent memory accesses. Therefore, dynamic instruction count is measured until each initialization program is terminated. While initialization programs produce the same result regardless of what is stored in the lookup table, packet processing programs produce different results depending on what is stored in the lookup table because the program runs through different paths. Therefore, we evaluate the packet processing programs by comparing each dynamic instruction count per iteration with some different lookup table layouts.

Initialization programs used in the experiment are fsp_1, fsp_2 and fsp_3, and packet processing programs are l2_processing and l3_processing.

5.2. Effectiveness of compiler techniques

To isolate the effectiveness of each technique for the benchmarks when being executed on a simulator, we conducted experiments following several different strategies, where each set of techniques was incrementally added to achieve better performance.

(1) **Baseline**: we measured the code size and dynamic instruction count of each code generated by the modified Zephyr without any optimization techniques except the intrinsic functions, which are indispensable to retarget Zephyr to PPII.

(2) **VDP (Virtual Data Path)**: to reduce the number of memory accesses inside each function, the virtual data path technique was added.

(3) **VDP+IRR (Interprocedural Register Renaming)**: strategy VDP was extended with the interprocedural register renaming to reduce the memory access over the entire program.

(4) **VDP+Inlining**: strategy VDP was extended with the function inlining instead of interprocedural register renaming.
Figure 20 shows the code size of machine codes\(^1\). These results show that strategies VDP, VDP+IRR, and VDP+Inlining are always smaller than strategy Baseline. This means that the techniques stated earlier can eliminate many memory access instructions. In the case of fsp_2 and fsp_3, strategy VDP+IRR is the same as strategy VDP because fsp_2 is a single function program and the callees of fsp_1 and fsp_3 do not have any local variables. So memory access instructions cannot be eliminated by interprocedural techniques such as interprocedural register renaming and function inlining. However in the case of l2_processing and l3_processing, strategy VDP+IRR is smaller than strategy VDP because interprocedural techniques eliminated the memory access instructions, which are needed when invoking a function. It is well known that the disadvantage of function inlining is increased code size \([17]\), but in the case of fsp_1 and l2_processing, the code size after inlining was decreased because most functions are invoked from a single call site and also because function inlining eliminated many memory access instruction to save/restore data registers and parameters passing overhead. In the case of l3_processing, the code size after inlining was significantly increased. This result is undesirable because l3_processing is a firmware program stored in the program memory.

---

\(^1\)These machine codes are generated from the compiler-generated codes by the PPII assembler to be executed on the PPII simulator.
whose size is limited. Therefore, in the aspect of code size, interprocedural register renaming is generally better than function inlining.

Figure 21 shows the dynamic instruction counts. One thing we can note from the results is that strategy VDP always does better than strategy Baseline. This means that many memory access instructions can be eliminated when the virtual data path technique is used. Taking into account that each memory instruction takes three cycles, the execution time gain is better than the dynamic instruction count gain in practice.

The following list provides a brief summary of each benchmark program and an analysis of the experimental results.

(1) *fsp_1*. This program has the pattern where three data values are read from the lookup table memory, summed up through the add function and then the result is saved into the lookup table memory. The add function simply accepts the parameters as operands and then returns the result of summation, so this function does not need any registers except parameter registers. Therefore, the results of strategies VDP and VDP+IRR are exactly the same. However, when strategy VDP+Inlining is applied, passing parameters and the return value via data registers are eliminated by function inlining. So better performance is achieved.
(2) \textit{fsp} \textsubscript{2}. This program initializes the lookup table entries to read each value from the SRAM location assigned to this program and store it to the lookup table memory in order. Because this code consists of a single function, the results of strategies \textit{VDP}, \textit{VDP}+\textit{IRR}, and \textit{VDP}+\textit{Inlining} are all the same.

(3) \textit{fsp} \textsubscript{3}. This code reads a sequence of data arrays from the lookup table memory, and then stores it into the space assigned in SRAM. In addition, this code performs addition or subtraction operations on data stored in SRAM, according to the value for initialization and then stores the result into SRAM again. The results of strategies \textit{VDP} and \textit{VDP}+\textit{IRR} are exactly the same and strategy \textit{VDP}+\textit{Inlining} achieves better performance than strategies \textit{VDP} and \textit{VDP}+\textit{IRR} for the same reason as \textit{fsp} \textsubscript{1}.

(4) \textit{l2-processing}. This is a program that processes a packet by means of Layer 2 information. First, it reads the source addresses (SAs) from the ingress buffer. If the SA is valid, then the destination address (DA) is also read from the buffer. If the DA is valid, then the packet is forwarded to the machine at the DA. Otherwise, there is no destination, so packet flooding is needed. Because whether each address is valid depends on contents of the lookup table, the execution path of the \textit{l2-processing} program varies according to the contents of a packet and the contents of a lookup table. Therefore, we created two different contents of the lookup table memory and then measured performance. The normal case is when both the SA and the DA in the packet are valid. In the normal case, only one function is executed and then this program is terminated. The packet flooding case is when the SA is valid but the DA is invalid. In this case, the packet flooding is performed since DA is invalid. That is, the function to handle the packet flooding error is invoked and the program is terminated. Therefore, in the normal case, because only one function is executed, all results are the same. But in the packet flooding case, strategies \textit{VDP}+\textit{IRR} and \textit{VDP}+\textit{Inlining} achieve better performance than strategy \textit{VDP}.

(5) \textit{l3-processing}. This program processes a packet by means of Layer 3 information. This program determines the route to the next router according to the destination IP address extracted from a packet. To do this, first the IP address is extracted from the packet. Then, the extracted IP address is compared with the routing table in order to find an optimal route. Unlike \textit{l2-processing}, \textit{l3-processing} contains function calls in the normal case. These functions are \texttt{FindEntry()} for finding the routing table entry with IP address and \texttt{L3PktModify()} for a packet update. Therefore, unlike \textit{l2-processing}, in the normal case, strategies \textit{VDP}+\textit{IRR} and \textit{VDP}+\textit{Inlining} achieve better performance than strategy \textit{VDP}.

6. RELATED WORK

As mentioned previously, the idea of NPUs is quite new and only a few products have been introduced in industry at the time this paper was written. Besides, most NPU vendors have developed in-house compilers whose technologies are only internally available. As a result, to the best of our knowledge, there are few research compilers for NPUs that have been described in the literature. In this section, we will discuss two such compilers.

The first compiler was developed at Informatik Centrum Dortmund (ICD) for Infineon Technologies network processor \cite{5,10,18}. Like most other compilers, this compiler is divided into a frontend and
a backend. It uses the LANCE compiler system [19] developed at the University of Dortmund as its frontend with a nearly fully customized backend. In the case of their target machine, Infineon NPU, the instruction set permits performing ALU computations on bit packets which are not aligned to 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.

This feature is called packet-level addressing. In order to enable packet-level addressing of unaligned data, the instruction set permits the specification of offsets and operand lengths within registers. Hence, as we did in this work, this compiler is focused on use of intrinsic functions to support the Infineon NPU instruction set. In addition, register files are called register arrays and to handle these register arrays, the authors devised a multi-level graph coloring algorithm [10], which is an extension of the classical graph coloring approach. However, they did not discuss how the heterogeneous data path and register files of NPUs are handled and did not describe the interprocedural techniques to reduce memory accesses.

The second NPU compiler [6] was built for STMicroelectronics’s network processor, which is designed to include various accelerating techniques such as hardware multi-threading, hardware accelerators for header parsing, checksums, and other intensive per-packet networking tasks. To best exploit the architecture, STMicroelectronics NPU’s embedded software tool set is based on the FlexWare embedded software development environment that has already been used within the company for a wide range of general-purpose and application-specific DSPs and micro-controller units. The FlexWare environment contains the C compiler called FlexCC, which is a retargetable C compiler. However, the authors did not report in detail how their compiler is organized.

7. CONCLUSIONS AND FUTURE WORK

In this work, a compiler for a commercial network processing unit is developed using the Zephyr compilation system, which has been developed mainly for GPPs and recently retargeted to DSPs. As compared to traditional GPPs with on-chip memory or cache, NPUs often require more aggressive compiler techniques to reduce loads/stores due mainly to their lack of on-chip memory or cache. In order to cope with this issue, we applied the virtual data path approach locally and some other conventional techniques, such as register renaming and function inlining, were applied interprocedurally. Experimental results indicate that these techniques collectively help us to improve the overall performance.

We believe that the contribution of this work is to explore the possibility of extending Zephyr, one of the traditional compilers developed mainly for conventional, yet popular, RISC-style GPPs, and retargeting it to more irregular NPU architectures, which has new features specialized to networking. Moreover, this contribution is to guide the optimization techniques for compilers targeting NPUs. Experimental results indicate that these techniques work well in practice, so that these techniques can be used commercially.

A number of interesting topics still remain open for future work. For example, we found that the function inlining achieves better real-time performance than interprocedural register renaming. But if passing parameters via registers is well designed, the interprocedural register renaming is able to achieve comparable performance as function inlining without any growth of code size.
ACKNOWLEDGEMENTS

We greatly thank Dr David Whalley at Florida State University, who is one of the original developers of Zephyr, for helping us to revise this paper including the part of our discussion about Zephyr. This work is supported in part by KRF contract D00263, ETRI, KOSEF R08-2003-000-10781-0, a seed grant for a new faculty member from SNU and also supported in part by NASA Idaho EPSCoR grant.

REFERENCES