An Interprocedural Code Optimization Technique for Network Processors Using Hardware Multi-Threading Support

Hanno Scharwaechter, Manuel Hohenauer,
Rainer Leupers, Gerd Ascheid, Heinrich Meyr
Integrated Signal Processing Systems
RWTH Aachen University
Aachen, Germany

Abstract

Sophisticated C compiler support for network processors (NPUs) is required to improve their usability and consequently, their acceptance in system design. Nonetheless, high-level code compilation always introduces overhead, regarding code size and performance compared to hand-written assembly code. This overhead results partially from high-level function calls that usually introduce memory accesses in order to save and reload register contents. A key feature of many NPU architectures is hardware multi-threading support, in the form of separate register files, for fast context switching between different application tasks. In this paper, a new NPU code optimization technique to use such HW contexts is presented that minimizes the overhead for saving and reloading register contents for function calls via the runtime stack. The feasibility and the performance gain of this technique are demonstrated for the Infineon Technologies PP32 NPU architecture and typical network application kernels.

1. Introduction

Network Processing Units (NPUs) are increasingly becoming popular for the utilization in modern *System on Chip* (SoC) design. They fill the gap of cost efficient yet flexible system solutions for evolving applications that allow packet processing at high data rates. Since the spectrum of network applications spans from traditional network access, realized by embedded software running on general-purpose architectures, up to core network applications, where high performance is required to meet line speed, NPUs exhibit a wide range of architectures: from simple general-purpose RISC cores with dedicated peripherals, in pipelined and/or parallel organization, to heterogeneous multiprocessors, based on complex *multi-threaded* cores with customized instructions [11, 10].

Multi-threading is a key feature to hide memory access latencies and therefore, to efficiently use the hardware of a NPU. It enables the architecture to process other streams while another thread is waiting for a memory access (or a different interrupt). Without hardware support, the cost of switching between different contexts (threads, processes etc.) would dominate computation time. Thus, NPUs support multiple hardware threads and register files to avoid storing and reloading the entire state of the machine during a context switch.

Due to the lack of good C compiler support for NPUs, assembly-level programming is currently state of the art, but using high-level languages gains more and more acceptance. However, high-level languages usually imply certain overhead in code size and performance compared to hand-written assembly code. While this overhead is acceptable for general-purpose computing, the demands on compilers are different for network processors. In particular, register allocation issues are very important since the latency of accessing off-chip DRAM is in tens of cycles and normally results in a context switch to another thread. One of the primary sources for the overhead introduced by high-level languages, are functions. Functions provide an appropriate technique to reduce code size and structure program code of complex modern applications by encapsulating repeatedly occurring portions of code. Saving and loading the functions' states, realized by several memory accesses, mainly causes the overhead introduced by function calls.

Function Inlining is a well-known technique used in many compilers for general-purpose processors which replaces function calls by copies of the related function's body. In this way, the function is turned into a high-level macro. Since the overhead associated with function calls (parameter passing, call and return instructions, saving and restoring register contents) is eliminated, function inlining tends to increase performance. However, function inlining also increases code size drastically and is therefore not always applicable for embedded system processors like NPUs due to their very limited program memory.

In this paper, a new technique to reduce the overhead of function calls is proposed that uses the HW multi-threading support of the NPU. The basic idea is that during code execution not all HW threads may be utilized at the same time by the tasks of a given application. Thus, the available free resources can be used by the compiler to optimize the code for the present tasks. Naturally, this technique requires compile-time knowledge of the processor's task load. For NPUs this information is usually at hand, since the use of operating systems and dynamic task creation are uncommon in network processing. By integrating the proposed technique into a generated C compiler [5] for the Infineon PP32 network processor [9], the feasibility and the performance gains are demonstrated.

3-9810801-0-6/DATE06 © 2006 EDAA

The technique exploits separate register files for function calls and thus, eliminates the necessity of storing and reloading register contents according to save the caller's state. However, due to the limited number of available register files, it is not possible to execute every function call with a new register file. Therefore, appropriate candidates have to be selected in order to maximize the benefit of this technique.

The remainder of this paper is organized as follows: In the next section, we describe related work. The following sections 3, 4 and 5 represent the core of our work. Here, first the driver architecture, Infineon's PP32, is described in section 3 with a subsequent introduction to calling conventions in general and a proposal of a "low overhead" calling convention for network processors in section 4. Finally, in section 5, we give an insight into the realization of the proposed calling convention in our compiler. This includes, on the one hand, an overview of our system and, on the other hand, an explanation of a heuristic algorithm for the selection of appropriate functions. In the last sections, the obtained results (sec. 6) and a conclusion (sec. 7) are given.

2. Related Work

There are several compiler research studies that have been conducted on the development of compilers for network processors. Wagner [6] has developed a compiler for Infineon's PP32 [9]. Its instruction set permits performing ALU computations on bit packets which are not aligned to the processors 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 addition, due to the feature of register files, they devised a *multi-level graph coloring* algorithm [6] which is an extension of the classical graph coloring approach. However, they did not discuss how heterogeneous data path and register files of NPUs are handled or interprocedural techniques to reduce memory accesses.

Based on Intel's IXP1200 architecture, two approaches on efficient code generation for network processors have been published in [12] and [13]. The IXP architecture contains two separate register banks, each of which is used to hold one of two operands of an ALU-instruction. Zhuang and Pande explain in [12] approaches to perform the operand assignment to a certain register bank either before, during or after the register allocation. In [13], he describes a way to allocate registers for multiple threads. The presented compiler analyses the register requirements of a thread, both at the time of a context switch and during the thread-execution. The registers are allocated accordingly either publicly accessible for all threads or privately for only one certain thread.

Finally, Paek presents in [8] several compiler optimizations while retargeting the Zephyr compiler system for the PaionII network processor. In the conclusion, he emphasizes the need for the reduction of memory access instructions. Similar to our work are approaches to exploit register windows of the SUN SPARC workstation with a compiler to reduce subroutine call overhead [3]. Wall describes in [3] several approaches to use registers on the SPARC architecture and compares the results. Whereas every compiler has to perform register allocation for its target architecture in order to produce efficiently running executables, our approach represents an optional way to exploit so far unused resources of a typical network processor after register allocation is already done.

3. Driver Architecture: Infineon PP32

The PP32 architecture is a 32-bit RISC-based *Application Specific Instruction Set Processor* (ASIP) [7] developed by Infineon Technologies (fig. 1), whose instruction set has been tailored towards the processing of network protocols like IPv4, IPv6 etc. The PP32 features several special instructions (e.g. bit-level instructions) as well as four *HW contexts* for fast task switches in multi-threaded applications, each comprising a separate register file. Every register file contains 16 *General Purpose Registers* (GPRs) as well as several *Special Purpose Registers* (SPRs) for *Program Counter* (PC) etc., such that each task keeps its own PC, identification (task1, task2 etc.) and also PC and task identification (oldPC, oldTask) for the program execution after its termination.

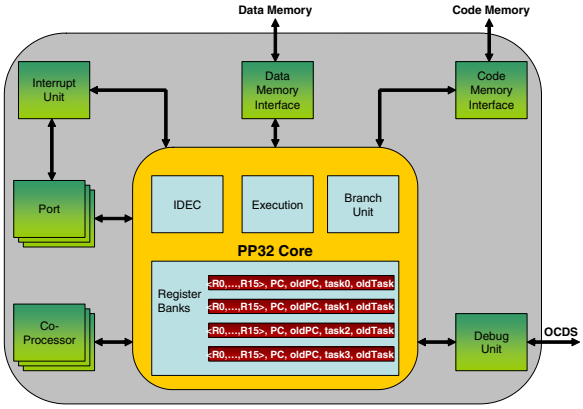

Figure 1. Architecture of the PP32

Our simulation model of the PP32 data path consists of a four-stage pipeline, I/O ports and supports instructions for

- data transfer (LDW, STW etc.),
- branch and thread control (BRREG, RET etc.), and
- logic + arithmetic (ADD, SUB etc.).

It also supports predicated execution determined by certain flags.

To implement a low overhead calling convention, we extensively used three dedicated hardware instructions which provide the base functionality:

RUNREG: The run-operation "RUNREG" starts a new task with a given task number and a given 32 bit branch address. It assigns the branch address to the PC and changes into the register file, indexed by the task number. At the same time, the old PC and task number are stored in additional SPRs which can be later used by "STOP" to return to the old task.

MVR2R: The move-operation "MVR2R" receives four parameters in registers: source register (RS) and task number (*src_taskno*), as well as destination register (RD) and task number (*des_taskno*). The operation transfers the value in RS of register file *src_taskno* into the register RD of register file *des_taskno*.

STOP: The stop-operation does not receive any parameters. It reads the return PC and task number from certain special purpose registers and performs a jump back to the old task.

4. A Low Overhead Calling Convention for Network Processors

The *calling convention* [2] is a contract between two functions – the *caller* and the *callee* – about the procedure of switching from the caller to the callee and back. Whereas the caller is responsible to pass the callee's function arguments in according registers, the callee has to conserve the caller's state, in order to guarantee the validity for the scopes of the local variables. This is established through the extension of each function by a *prologue* and an *epilogue*. These two code paragraphs enframe the *function body* of every function and together they perform the callee's part of a calling convention. The job of prologue and epilogue inside a calling convention is to save (prologue) and reload (epilogue) the caller's state which is represented by the actual register values before the caller's function call of the callee. Figure 2 gives an example of a typical traditional calling convention.

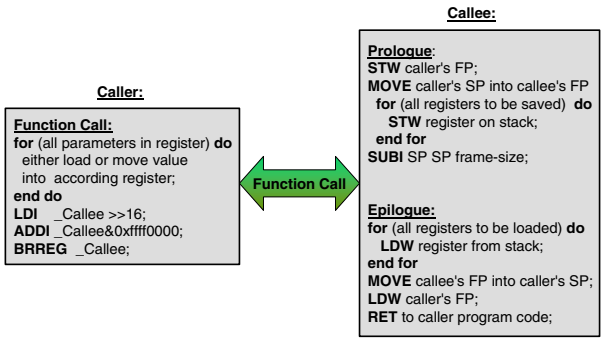

Figure 2. Traditional Calling Convention

After the caller has placed necessary function arguments in according registers and executed a call instruction, in the callee's prologue (fig. 2), first the caller's *Frame Pointer* (FP) is saved and afterwards, the caller's *Stack Pointer* (SP) is moved into the callee's FP. Subsequently, all register values that represent the caller's state have to be saved onto the stack and the new SP for the callee is computed. Stack Pointer and Frame Pointer of a function are special register values that denote the base addresses, used for accessing local variables and parameters.

In the epilogue of figure 2, first all register values that represent the caller's state have to be reloaded from the memory. After this, the callee's FP is moved into the caller's SP and the caller's FP is also reloaded from the memory.

Figure 3 shows the exact procedure of the calling convention using a separate HW context for the callee. Here, the

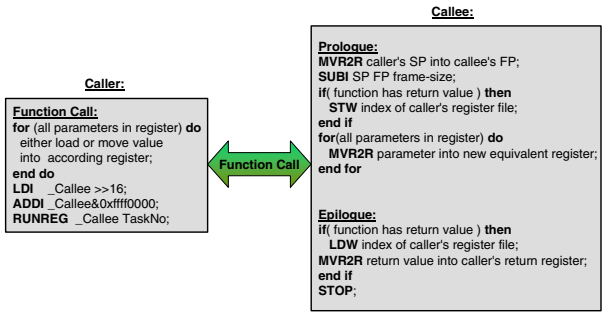

Figure 3. Low Overhead Calling Convention

caller uses "RUNREG" to evoke the callee. In the callee's prologue of figure 3, first the caller's SP is moved into the callee's FP and the new SP is computed by subtracting the current frame size from the FP. Now, in case of a return value, the index of the register file has to be saved for later utilization in the epilogue. In a final step, all parameters residing in registers are moved to the HW context.

In the epilogue of figure 3 just the return value – if necessary – is moved to the caller's register file and the callee's task (function) is stopped. Consequently, no LOAD/STORE-instructions have to be executed in order to save/reload the caller's state.

The trade-off in choosing between these two calling conventions relies on the quantitative relation of parameters and memory accesses, necessary to save and reload the caller's state. That is, if the number of parameters is less than the amount of necessary memory accesses, the low overhead calling convention will introduce less overhead than the traditional one, but vice versa, if there are more parameters to be transfered between the register files than memory accesses are necessary to save and reload the caller's state, then the traditional calling convention will be more advantageous.

5. Optimized Selection of Calling Conventions

Due to the limited number of available HW contexts (four in case of the Infineon PP32 NPU), obviously not every function can be executed in a separate HW context. As a consequence, appropriate candidate functions have to be found by the compiler, such that the benefit of using separate HW contexts for function calls is maximized. In order to evaluate every function's quality according to the previously described calling conventions (section 4), a metric has been established which can be used to sort functions and to decide for each function which convention is best to be applied. Based on this metric, the compiler is able to select for each path in the call-graph of the source application, the best candidates that are worth to be executed in a separate HW context.

5.1. System Overview

Figure 4 presents a complete system overview of our compiler framework that has been used to implement a low overhead calling convention.

The simulation model of the Infineon PP32 (section 3) has been developed with the LISATek Processor Designer

[4]. Therefore, we have been able to automatically generate the simulator, assembler, linker, and semi-automatically a C-compiler [5] from the LISA-model to execute typical network applications on this platform.

Our compiler for the PP32 is based on the CoSy Compiler Development System from ACE [1]. CoSy is a highly modular C/C++ compiler framework, consisting of loosely coupled *engines*, working on a central *Intermediate Representation* (IR). The retargetable backend engine is based on a *Code Generator Description* (CGD). The CGD enables developers – amongst others – to specify available *target* processor resources like registers or functional units and to describe a set of *mapping rules*, determining how C/C++ pattern match (potentially blocks of) assembly instructions. Those mapping rules, specifying the prologue, epilogue and function calls have been used to modify the calling convention.

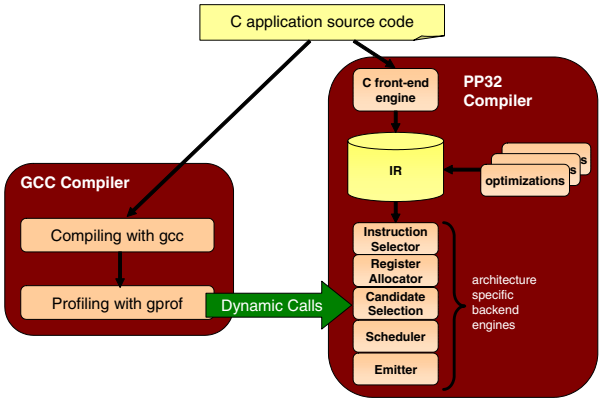

Figure 4. System Overview

The algorithm for candidate selection is implemented as a single engine that has been inserted into the backend of our PP32 compiler. Since the algorithm needs special register information, the engine is executed after the register allocator. Furthermore, runtime information is required to determine the number of dynamic calls for each function. To provide this information, the source application is profiled in advance by the freely available GNU profiler *gprof*. Gprof stores the obtained runtime information in files, such that the number of dynamic function calls can later be easily accessed by the PP32 compiler.

5.2. Candidate Selection

For a given application C source code, our technique requires the following input data

- A **static call-graph** $G = (V, E)$. G is a *Directed Graph*, where each node $v \in V$ represents a function and each edge $(v_i, v_j) \in E$ depicts a call dependency from v_i to v_j. In order to avoid infinite loops, recursive functions and functions with a call cycle have to be excluded from the algorithm. Also top-level function, i.e the "main" function, or functions not called anywhere in the source code are not considered as candidates by the algorithm. Furthermore, functions without a body, i.e. standard library functions like "printf", are not taken into account.
- The number $P(f)$ of **parameters** residing in registers for each function f.
- The number $R(f)$ of **registers to be saved and reloaded** by traditional calling convention for each function f.
- The number N of **HW contexts** available in the target architecture.
- The number $D(f)$ of **dynamic calls** for each function f. This information is obtained by profiling.

For candidate selection, one has to consider all paths $P = (p_1, \ldots, p_n)$ of G starting at the root p_1 that corresponds to the "main" function in a C program. As one HW context is always occupied by "main", due to its liveness throughout program execution, $N - 1$ nodes (instead of N) have to be identified within every path p_i whose implementation by the low overhead calling convention explained in section 4 leads to the highest gains in code quality.

Let Q denote the set of all sub-sets $q = (f_1, \ldots, f_{N-1})$ with length $N - 1$ of a given path p_i. That is, each f_i in q corresponds to a particular function along a call graph path. For a sub-set q we define the benefit $B(q)$ as

$$B(q) = \sum_{\forall f \in q} (R(f) - P(f))D(f).$$

$B(q)$ measures the cost savings as the difference of registers to be stored and loaded (traditional calling convention) and the number of register parameters (low overhead calling convention), scaled by the number of dynamic calls of function f. The best selection of candidates obviously corresponds to determining the optimal sub-set $q* \in Q$ such that $B(q*)$ is maximal among all $q \in Q$.

Our heuristic algorithm recursively traverses the call-graph G in depth-first order, starting at the root, and identifies possible function candidates for being executed within separate HW contexts. The recursion is terminated at the leaf nodes/functions which do not contain any function calls.

The strategy applied by depth-first traversal is, as its name implies, to traverse "deeper" in the call-graph, whenever possible. In depth-first traversal, edges are explored out of the most recently discovered vertex v that still has unexplored edges emanating it. When all of v's edges have been explored, the traversal "backtracks" to explore edges leaving the vertex from which v was discovered. This process continues until we have discovered all the vertices that are reachable from the original source vertex. If any undiscovered vertices remain, then one of them is selected as a new source and the traversal is repeated from that source. The entire process is repeated until all vertices have been traversed.

Figure 5 presents the heuristic candidate selection in pseudo code. An essential part of the algorithm is the *sorted_cands* input. *Sorted_cands* is an array that keeps up to $N - 1$ function nodes in ascending order of their benefits B. In case of an overflow, the node with lowest benefit B is excluded from the array and the remaining nodes are ordered by their benefits. Using the *sorted_cands* array, the candidate selection takes place in two phases for each node. Firstly, the node's benefit is computed and, if positive, inserted into *sorted_cands*. While traversing deeper, the node's adjacency list is examined and consequently, *sorted_cands*

```
algorithm depth_first_traversal
input: Graph G = (V, E),
       Node v ∈ V,
       sorted_cands[1 ... N-1];
output: Annotated graph G+ = (V+, E)
begin
01  f = v;
02  if (f is not recursive) then
03     if (B(f) > 0) then
04        Sort f into sorted_cands ;
05        Assign f to a register file;
06     end if
07  end if
08  for ( all callees of f ) do
09     depth_first_traversal( G, callee, &sorted_cands );
10  end for
11  if ( f in sorted_cands ) then
12     delete f from sorted_cands ;
13     delete assignment of register file;
14     annotate selection of f in G;
15  end if
```

Figure 5. Pseudo code of the selection algorithm

keeps for each node the $N-1$ best previously selected candidates. Secondly, if the node's adjacency list has been entirely examined, and the node is an element of *sorted_cands*, it is removed from the array and finally annotated in the call-graph G as being executed in a separate HW context.

The algorithm results in an annotated call-graph $G^+ = (V^+, E)$, where V^+ designates the set of nodes V including a subset of marked nodes that represent the selected candidates for separate HW contexts. Since the IR of CoSy is a graph, the call-graph is a subset of the IR. Therefore, the information about selected candidates is available for succeeding compiler-phases like the code-emitter which produces the assembly code for calling conventions as described in section 4.

5.3. Example

Consider the static call-graph of figure 6, with 4 function nodes and corresponding benefits $B(f)$, annotated for each node f in the graph. The corresponding traversal of the graph is presented on the right hand-side of figure 6, where the nodes present the appropriate states of the *sorted_cands* array (fig. 5). Furthermore, the according number of available HW contexts N is 3, such that the number of candidates is 2, because one register file is always occupied by "main".

The algorithm will start at the root "main", take the first available callee F1, insert it into *sorted_cands*, proceed to F2 and insert this node as well into *sorted_cands*. Arriving at node F3, both slots of *sorted_cands* are occupied with predecessors of F3, such that the "weakest" node F2 will be excluded while sorting F3 into *sorted_cands*. Since F3 has an empty adjacency list, no further callees have to be visited on this path. The algorithm tracks back to a predecessor with a non-empty adjacency list and eliminates the passed nodes from *sorted_cands*. These functions are at the same time selected for the execution in a separate HW context. The corresponding nodes in the graph traversal are marked dark to emphasize the final selection of a function by the algorithm.

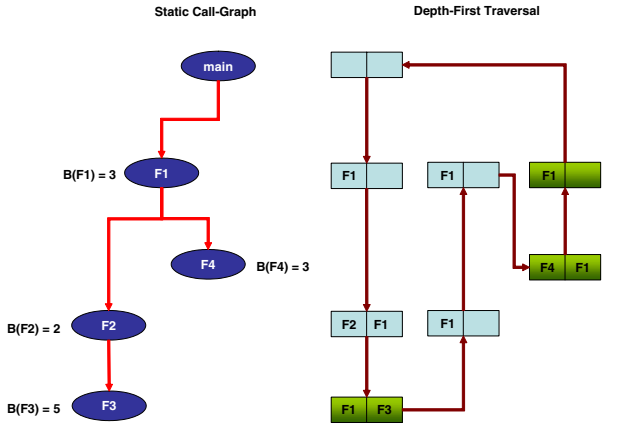

Figure 6. Example call-graph and traversal with corresponding states of sorted_cands

5.4. Algorithm Complexity

We have chosen this depth-first approach, due to its ability to compute an optimized solution in a short runtime: Lines 1 - 7 and lines 11 - 15 of figure 5 take time $O(V)$, excluding the time to execute the recursive calls for the adjacency list of the actual vertex v in lines 8 - 10 of figure 5. The loop in lines 8 - 10 is executed $|Adj[v]|$ times, because it is called for every callee of the actual node v. Consequently, $\sum_{v \in V} |Adj[v]| = O(E)$ and therefore, the total cost of the algorithm is $O(V+E)$. Since the "MVR2R" instruction (section 3) receives all of its operands in registers, we can dynamically determine for each function call, in which HW context it will be executed. Therefore, no global dependencies between function calls inside different paths of the call-graph exist.

6. Experimental Results

In order to evaluate the proposed technique, application studies for several typical network applications provided by Infineon, have been performed. The benchmark suite comprises an IPv6 Router, an Ethernet Router and a test program for the signal ports. As presented in table 1, the benchmarks contain between 1180 and 2223 lines of code. The number of functions and the quantitative portion of selected functions are given in the following rows.

Table 1 presents our simulation results for the network applications in the following section. All results have been obtained from the same compiler, once with enabled and once

with disabled candidate selection. The results represent the relative speedup of the optimization given in percentage. As with Function Inlining, our optimization relies strongly on the application's paritioning of functions. Consequently, both optimzations are orthogonal and cannot be executed independently, such that Function Inlining has been switched off all the time, in order to get more reliable results by examining a bigger set of functions. The obtained results are therefore relative values based on the available set of functions.

	Results of Benchmarks			
	IPv6 Router	**Ethernet Router**	**Port Access**	**Average**
lines of code	2075	1180	2223	–
functions	31	28	29	–
sel. functions	26	25	21	–
speedup (1)	13.1%	9.7%	16.6%	13.1%
speedup (2)	17.5%	13.0%	21.5%	17.3%
speedup (3)	20.7%	15.5%	25.0%	20.4%
speedup (5)	24.9%	18.8%	29.3%	24.3%
speedup(10)	30.2%	23.1%	34.6%	29.3%
code size	-2.7%	-2.2%	-2.0%	-2.3%
LOAD	-36.6%	-33.8%	-41.3%	-33.9%
STORE	-43.1%	-36.0%	-42.8%	-40.6%

Table 1. Overview of experimental results

The values have been obtained for different configurations of the memory's wait cycles. Assuming that apart from ideal memories, every memory produces at least 1 wait cycle per access, the memory model has been configured for wait cycles between 1 and 10 which are given in parentheses for each row. Even for an extremely fast memory (1 wait cycle), a significant speedup (13.1% on average) has been measured. Naturally, the speedup grows with more realistic wait cycle count (e.g. up to 29.3% for 10 wait cycles).

A secondary optimization effect is an average code size reduction of 2.2%. This is due to the lower number of instructions needed for context switching. Hence, as compared to a related interprocedural optimization (function inlining), the speedup has not to be paid by an increase in code size.

In the last section of table 1, the relative reduction of dynamic memory accesses for all benchmarks is highlighted. LOAD instructions have been reduced by 33.9% and STORE instructions by 40.6% on average. We believe that especially these results will also affect the power consumption of a network processor, because memory accesses usually belong to the most power consuming hardware instructions.

7. Conclusion

Many NPUs are equipped with HW multi-threading support by means of different HW contexts. This paper presents a novel compiler optimization that exploits HW contexts not fully utilized by the tasks of an application. It attempts to reduce the overhead of high-level function calls which largely results from memory accesses in the prologue and epilogue of each function. The technique has been implemented into a C compiler for the Infineon PP32 NPU and has been successfully tested for different typical NPU applications.

The proposed code optimization is very effective as it leads to a significant speedup of the executables, as well as to a small code size reduction as a secondary effect. Although we have no experimental confirmation as yet, it is anticipated that a significant saving in power consumption results as well, due to the large reduction of memory accesses via LOAD and STORE instructions.

The implementation proposed in this paper is mainly targeted for the PP32 architecture. However, we believe that retargeting for further NPU architectures with HW multithreading support is straightforward. In the future, we will investigate such retargeting mechanisms, as well as further NPU-specific code optimizations in the PP32 C compiler.

References

[1] ACE – Associated Computer Experts bv. *The COSY Compiler Development System http://www.ace.nl*.

[2] A. Aho, R. Sethi, and J. Ullman. *Compilers, Principles, Techniques and Tools*. Addison-Wesley, 1986.

[3] D. W. Wall. Register Windows vs. Register Allocation. In *Conf. on Programming Language Design and Implementation (PLDI)*, pages 67–78, June 1988.

[4] A. Hoffmann and H. M. et al. A Novel Methodology for the Design of Application Specific Instruction Set Processors (ASIP) Using a Machine Description Language. *IEEE Transactions on Computer-Aided Design*, 20(11):1338–1354, Nov. 2001.

[5] M. Hohenauer, H. Scharwaechter, K. Karuri, O. Wahlen, T. Kogel, R. Leupers, G. Ascheid, and H. Meyr. A Methodology and Tool Suite for C Compiler Generation from ADL Models. In *Proc. of the Conference on Design, Automation & Test in Europe (DATE)*, Mar. 2004.

[6] J.Wagner and R. Leupers. C Compiler Design for a Network Processor. *IEEE Trans. on Computer-Aided Design (TCAD)*, 20(11):1302–1308, Nov. 2001.

[7] K. Keutzer and H. M. et. al. *Building ASIPs: The Mescal Methodology*. Springer, June 2005. ISBN: 0-387-26057-9.

[8] J. Kim, S. Jung, Y. Paek, and G.-R. Uh. Experience with a Retargetable Compiler for a Commercial Network Processor. In *Proc. of the Conference on Compilers, Architectures and Synthesis for Embedded Systems (CASES)*, Oct. 2002.

[9] X. Nie, L. Gazsi, F. Engel, and G. Fettweis. A new network processor architecture for high-speed communications. In *Proc. of the IEEE Workshop on Signal Processing Systems (SIPS)*, pages 548–557, Oct. 1999.

[10] N. Shah. Understanding Network Processors. Technical Report 1.0, University of California, Berkeley, Sep. 2001.

[11] N. Shah and K. Keutzer. Network Processors: Origin of Species. In *The 17th International Symposium of Computer and Information Science*, 2002.

[12] X. Zhuang and S. Pande. Resolving Register Bank Conflicts for a Network Processor. In *IEEE International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2003.

[13] X. Zhuang and S. Pande. Balancing Register Allocation Across Threads for a Multithreaded Network Processor. In *Conference on Programming Language Design and Implementation (PLDI)*, June 2004.