In Proceedings of PARELEC 2004, International Conference on Parallel Computing in Electrical Engineering, Dresden, Germany, 7.-10. September 2004, p. 209-214.

Network Application Driven Instruction Set Extensions for Embedded Processing Clusters

Matthias Grünewald[*], Dinh Khoi Le[†], Uwe Kastens[†], Jörg-Christian Niemann[*], Mario Porrmann[*], Ulrich Rückert[*], Adrian Slowik[†], and Michael Thies[†]

Abstract

This paper addresses the design automation of instruction set extensions for application-specific processors with emphasis on network processing. Within this domain, increasing performance demands and the ongoing development of network protocols both call for flexible and performance-optimized processors. Our approach represents a holistic methodology for the extension and optimization of a processor's instruction set. The starting point is a concise yet powerful processor abstraction, which is well suited to automatically generate the important parts of a compiler backend and cycle-accurate simulator so that domain-characteristic benchmarks can be analyzed for frequently occurring instruction pairs. These instruction pairs are promising candidates for the extension of the instruction set by means of super-instructions. Provided that a new super-instruction meets a given performance threshold, a fine-grained performance re-evaluation of the adapted processor design can be conducted instantly. With respect to the chosen domain-characteristic benchmark, the tool-chain pinpoints important characteristics such as execution performance, energy consumption, or chip area of the extended design. Using this holistic design methodology, we are able to judge a refinement of the processor rapidly.

1. Introduction

The extension of instruction sets is typically driven by optimization goals that aim at speed-up, reduced code-size, reduced power consumption, and the like [1]. As the general-purpose core we start with has already proven to be efficient [2], it is mandatory to exploit specifics of a restricted application domain. Since application behaviour may heavily depend on problem input size and application context, we use feedback driven optimization in combination with application software that has been targeted for the specific application domain.

In the GigaNetIC project [3], we aim at developing high-speed components for networking applications based on massively parallel architectures. A central part of this project is the design, evaluation, and realization of a parameterizable network processing unit. The proposed architecture is based on embedded processing clusters, which are arranged in a hierarchical system topology with a powerful communication infrastructure (cf. Fig. 1). On the cluster level, four processing elements are connected via an on-chip bus to a so-called switch box, which allows to realize arbitrary on-chip topologies. Hardware accelerators can be added to trade flexibility for throughput, and they help to reduce energy consumption. Following a top-down approach, network applications are analyzed and partitioned into smaller tasks. The tasks are mapped to dedicated parts of the system, where a parallelizing compiler exploits inherent instruction level parallelism. Therefore, the hardware has to be optimized for this programming model in several ways. For example, synchronization primitives have to be provided, and memory resources have to be managed carefully. As a core component of our architecture, we use a 32 bit RISC CPU, the S-Core, which has already been designed and manufactured as a stand-alone processor [2]. In the following, we are focussing on the optimization of this core processing element and its migration to a more specialized network processing engine called N-Core.

In order to achieve significant improvements in terms of performance, code size, and power consumption, we propose to evaluate envisioned instruction set extensions by means of workloads that are representative for the targeted deployment scenario and cycle accurate simulation.

Methodology

The determination of super-instructions is divided into two steps. The first step is based on a compiler-related workflow, in which the application is analyzed with respect to the occurrence of instruction pairs. Frequently

[*] Heinz Nixdorf Institute and Institute of Electrical Engineering and Information Technology, University of Paderborn, Germany, {gruenewa,niemann,porrmann,rueckert}@hni.upb.de

[†] University of Paderborn, Germany, {uwe,le,adrian,mthies}@upb.de

used pairs are extracted and modeled as super-instructions. This first part of our workflow comprises the exploration of possible candidates for super-instructions. Due to the high abstraction level of the simulation, the exploration can be iterated very fast, thus allowing a broad search on the instruction space. The second step in our framework is based on real hardware implementations, thus consuming much more simulation time. Therefore, a detailed evaluation of instructions in the first step assures that only the most promising candidates are implemented and analyzed.

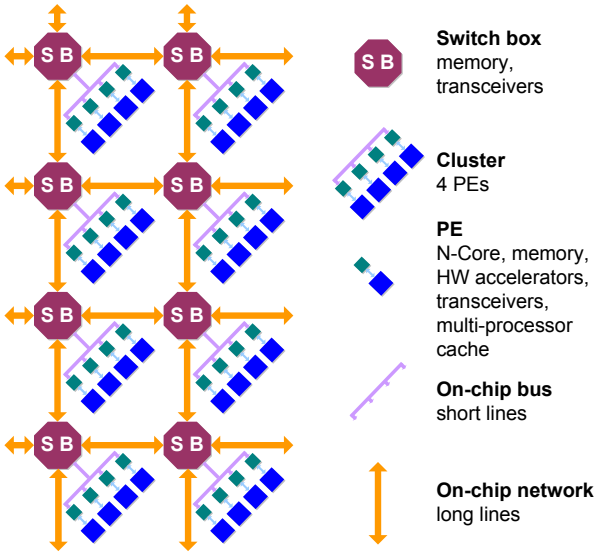

Fig. 1: System architecture based on embedded processing clusters

The first step of our framework is supported by a powerful tool chain that allows the designer to easily extract relevant combinations of instructions which will be defined as super-instructions. Then, the compiler is made aware of these super-instructions and automatically utilizes them during the next compilation. Finally, the impact of these modifications on system performance is analyzed with our tool chain.

If the performance achieved with our modified processor satisfies our needs we proceed to the second step of our workflow. In this step, the binary opcode definition is given to the hardware designer. He has to implement the new instruction in hardware by modifying the instruction decoder and by adding the necessary logic. The new design is synthesized to the corresponding technology. Relevant parameters like area and performance are extracted. If the design constraints are not met, a refinement of the hardware description is necessary. After fulfilling the requirements, a detailed performance analysis is carried out with our analysis framework PERFMON [4]. During this phase, several important parameters of the design are measured while running the desired application. These results are then compared to those of the unmodified instruction set of the processor. This enables us to judge the impact of the new instruction set on a given application with respect to required clock cycles, area consumption, and power consumption.

Our approach provides a holistic framework for optimizing and benchmarking a given instruction set for a given application. This enables us to provide a resource-efficient and optimized embedded system in conjunction with a super-instruction-aware compiler.

2. Compiler-related workflow

We rely on an iterative process to refine the processor and the corresponding software development tools. This process is initialized with benchmarks that are characteristic for the application domain and a formal model of the general purpose RISC-core. Fig. 2 provides an overview of this process. It starts with a formal model of the unmodified processor. This model is sufficiently expressive and precise to generate a C compiler and cycle-accurate simulator for the processor, but omits the unnecessary details of a full hardware design.

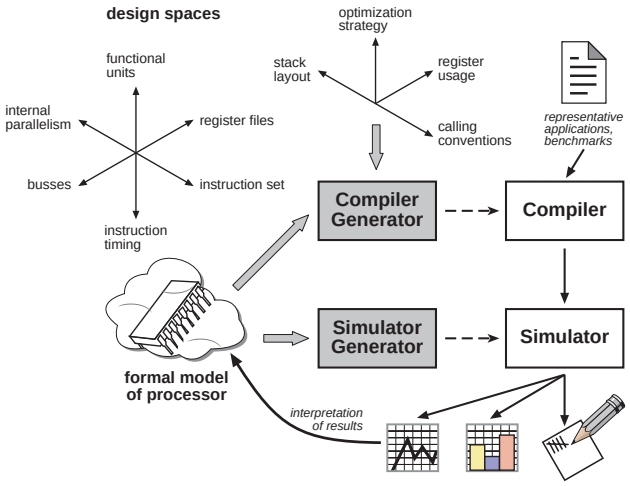

Fig. 2 Workflow of the compiler-related framework

Then, the generated development tools are used to compile benchmark programs and to simulate their execution on the envisioned processor. The simulator collects a wealth of statistical and performance data which is condensed and summarized visually for simpler comprehension. These results provide feedback for refinements of the processor architecture. The formal model is modified

accordingly, and the cycle starts anew. Processor variants are adopted, selected for further improvement, or rejected immediately based on the predicted performance of the benchmark programs.

This workflow allows to explore a design space which is spanned by two distinct kinds of parameters: hardware parameters and compiler parameters. The processor hardware is characterized by the size and the number of register files as well as the number and capabilities of functional units. Different compilers for the same processor hardware can employ different calling conventions, follow different register usage policies, or select a different ensemble of code optimizations. Both kinds of design parameters in concert determine the resulting perceived processor performance. If the design space is restricted to a reasonable number of parameters with rather limited domains, exploration can even be fully automated [5]. A branch-and-bound algorithm traverses the relevant part of the design space and records all Pareto points without human intervention.

Additional instructions combine the effects of multiple existing instructions, but execute faster. An exhaustive search through all pairs, triples, quadruples of instructions is neither reasonable nor feasible. Instead, our approach relies on hardware design tools driven by an expert to select effective combinations of instructions. Our simulator determines the most promising candidates, i.e., those that offer the highest performance gain. A dedicated visualization tool ranks and proposes these candidates to the human expert.

We use an augmented version of the generator `burg` [6] to build the code selection phase of our compiler. Code selection is specified by a tree grammar, where rules map fragments of intermediate language (IL) trees to sequences of target code. Each rule is annotated with the execution time of the attached target code. Then the code selector is able to cover each tree with minimum total costs in linear time.

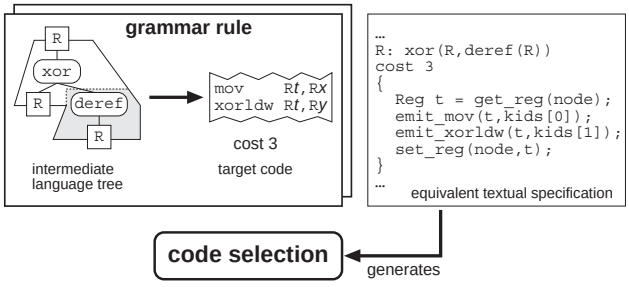

Fig. 3: Specification of the combined `xorldw` instruction

Fig. 3 shows both, a grammar rule for the new `xorldw` instruction and the corresponding part of the actual processor specification. The `xorldw` instruction computes the exclusive or of its target register and a word from memory. To map the 3-address semantics of the IL tree onto the 2-address semantics of the instruction, a temporary register and an additional `mov` instruction are needed. Register allocation eliminates this overhead, where permissible. The textual specification contains a fragment of C code that emits the corresponding instructions of the target code and manages any temporary registers required. Within the C code the predefined variable `node` refers to the root node of the matching IL tree fragment. The array `kids` refers to the root nodes of the sub-trees below the current tree fragment.

When the constituents of a new combined instruction have been determined, additional tools help to modify the formal model of the processor accordingly. In the case of the `xorldw` instruction a new grammar rule is created that combines the tree fragments of the constituent IL operations `xor` and `deref`. After the human expert has selected existing rules for the constituents, a rule template for the combined instruction is generated by the system. Next, the template is refined by the expert. This step requires some further input provided by the hardware tool chain, for example the execution time of the combined instruction. Finally, the system generates an enhanced compiler from the augmented specification.

Due to the extensive tool support, performance evaluation based on the extended instruction set can commence within a couple of minutes. The iterative refinement of the instruction set stops if the available opcode space has been exhausted, or, if the performance improvements of additional super-instructions fall below a certain threshold.

2.1 Simulator performance

We have conducted our experiments with the generated simulator on an Intel Pentium 4 CPU Linux platform running at 2.4GHz. According to the simulation performance observed, the simulated N-Core processor achieves 7.63MHz for the IPSec benchmark implementation we have used. Thus an Intel Pentium 4 CPU running at 2.4GHz needs approximately 313 clock cycles to simulate a single clock cycle of the N-Core processor. This is not the best performance our simulator can achieve, because the simulator also has to collect a wealth of statistics. Disabling these rich statistics provides a significant performance boost.

To verify the accuracy of our generated simulator we compare the collected performance figures with those of a VHDL-simulator, as detailed below. The VHDL simula-

tion at the gate-level takes substantially longer than the cycle-accurate instruction-level simulation (about 5 orders of magnitude, cf. section 4.2).

3. Hardware-related workflow

In [4], we have presented a simulation-based method to measure the (dynamic) execution time, energy consumption, memory accesses and stack usage of embedded software on synthesizable application-specific processing engines. In this context, our PERFMON framework has been used to analyze hardware accelerators for network applications such as cyclic redundancy checkers and random number generators in comparison to software implementations.

In this paper, our analysis framework PERFMON is used to analyze the impact of instruction set modifications on performance and resource efficiency for special network applications. Any processor for which a VHDL or Verilog model is available can be integrated into our environment. We have developed the N-Core, a processor core described in VHDL, allowing arbitrary modifications of the instruction set and architecture. The N-Core is a simple scalar, 2-address RISC CPU with a three-stage pipeline and an addressing interface that supports byte granular access. Moreover, it has 16 bit instructions and exhibits a high code density. The processor is well suited for the implementation of network protocols such as media access, routing or packet classification in embedded systems. On the one hand, the architecture can be expanded by adding coprocessors to the core. On the other hand it can easily be expanded by adding application-specific instructions due to 11% of free opcode space.

Our workflow (cf. Fig. 4) allows an analysis of arbitrary hardware accelerators (e.g., new instructions or coprocessors) which have been added to the processing engine. In our approach, the given C sources of the application that is to be optimized are compiled by the super-instruction-aware compiler. Finally, we receive a binary memory image that is stored into the RAM of our simulation environment. In the following gate-level synthesis with the SYNOPSYS Design Compiler, the hardware is mapped to a standard cell technology (UMC 130nm). The result is a database of the required gates and their interconnects. The next step is a gate-level simulation of our system with Cadence NCSim. During this phase all switching activities of the processor that runs the application code are being recorded.

Using the characterization files from the hardware vendor, a detailed power model of the processor is created with the SYNOPSYS Power Compiler. Annotating this power model with previously obtained switching activities leads to a power estimation of the processor for the given application. This estimation is evaluated and gives detailed information about the change in power consumption and the additional area needed for the supplemental instructions or coprocessors. In the case of unsatisfying results, an HDL description has to be refined and a new iteration of the tool chain has to be started. In section 4, we describe the results obtained with our workflow for a super-instruction needed in networking applications.

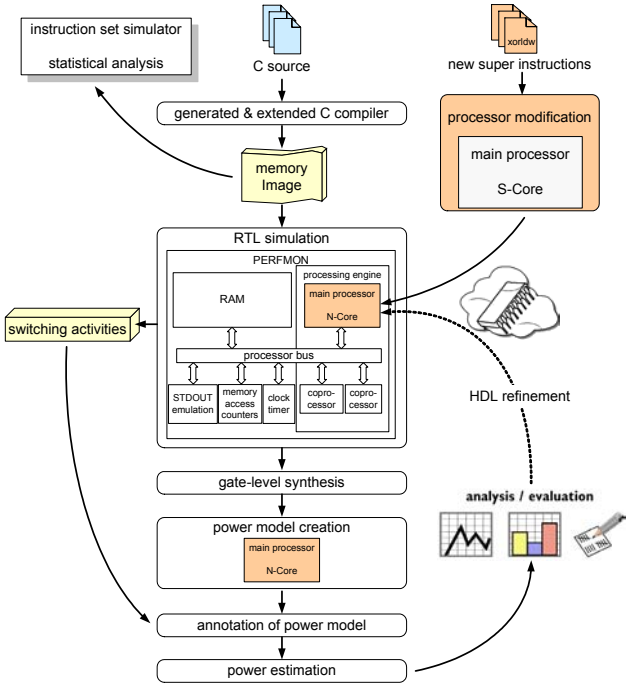

Fig. 4: Workflow of the hardware-related framework

4. Application-specific case study

We illustrate the effect of our approach with the example of a network processor dedicated to symmetric ciphers, which are the major building blocks of the IPSec security protocol framework. We compare the performance of the original and that of the extended instruction set. Within the first iteration of our approach, the general-purpose instruction set gets extended by an `xorldw` super-instruction, which fetches one operand from memory and applies an `xor`-operation to a second register operand, overwriting this second register.

4.1 Cycle-accurate simulation

Fig. 5 depicts the impact of the introduced `xorldw` instruction on the 3DES-CBC implementation taken from the open-ssl library 0.97 (triple DES with cipher block chaining):

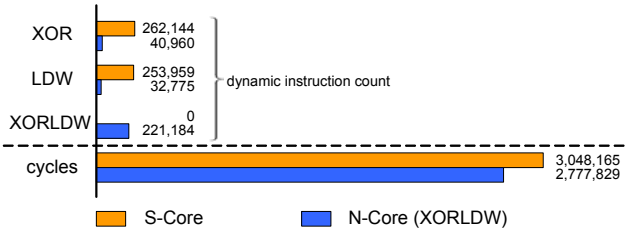

Fig. 5: Cycle counts of DES code before and after extension

The total number of cycles for encryption, measured with the cycle-accurate instruction-level simulator, drops from 3,048,165 cycles to 2,777,829 cycles, if the code is applied to a traffic pattern that adheres to the iMix packet size distribution. This gives a reduction to 91.1% of the original execution time. Please note also the new super-instructions affect code-selection for other code regions, i.e., the impact of a new super-instruction spans multiple instruction statistics.

4.2 Detailed hardware analysis

The implementation of the `xorldw` super-instruction was realized by modifying the instruction decoder (ID). The `xor` operation was already available in the existing logic of the ALU so that it sufficed to enable this operation by the ID in this new context. This kind of new instruction requires the smallest number of changes of the hardware by making use of existing combinatorial and non-combinatorial logic. It may increase the area and, in some cases, the critical path of the ID (cf. Table 1) and thus should be critically observed to assure that the impact on clock frequency and area is minimal. A more demanding modification of the design is the implementation of instructions that need additional arithmetic or logical units. This enlarges the design effort and requires changes that extend beyond the ID. The most time-consuming and area-demanding process is the implementation of hardware accelerators for even more complex tasks.

Microbenchmark

First, we have tested the new processor design with a microbenchmark with which the functionality and the resource requirements have been analyzed. In the second step, we applied the 3DES algorithm on both designs (cf. Fig. 6). The addition of the `xorldw` operation causes an increase of the ID area of about 1.3%. This equals 98.52µm², i.e., less than 0.08% of the total area needed for the processor. We have measured the power consumption of the original CPU for the execution of the two instructions `ldw` and `xor`. This pair is replaced by the `xorldw` instruction in the new processor design. The average power consumption was 8.875mW for the original processor and 8.781mW for the modified processor, which uses the super-instruction (considering a clock frequency of 205MHz in both cases). This is a decrease of 1.1%. Taking into account that the new instruction takes only two instead of three clock cycles, the impact on the total energy is significant. By using the new instruction, we observe an energy reduction of about 34% and a time saving of one third without a reduction of the clock rate. Furthermore, we save 50% instruction memory when using the new operation instead of the two traditional instructions.

Table 1: System parameters of the original and extended processor

	total area cpu		total area instruction decoder	
	absolute (µm²)	*relative increase*	*absolute (µm²)*	*relative increase*
S-Core	169675.83	0.0000%	7720.88	0.0000%
N-Core (XORLDW)	169801.91	0.0743%	7819.40	1.2599%

A closer look at the differences between the two designs makes it obvious that the ID and the register file as main components of the CPU core have the biggest impact on the savings in the overall power consumption. The ID of the new design needs 11.8% less power than that of the original processor. This goes hand in hand with a slight decrease of the power consumption of the register file, caused by fewer register accesses. In contrast to this, the power consumption of the ALU increases by 3.2%.

Performance improvements for 3DES

The simulation of the given DES application reflects an effective clock frequency of the simulation of the N-Core of about 150Hz. The whole hardware-related simulation toolflow takes about 7 hours on a 2.2GHz Pentium 4 CPU with 512MB RAM. Compared to the 7.63MHz of the cycle-accurate instruction set simulator (cf. section 2.1), this number emphasizes the necessity of our two-level approach.

Following the microbenchmark analysis of the two designs, we have benchmarked the complete 3DES applica-

tion on both cores. Fig. 6 depicts the total power consumption of both cores and their main components for the microbenchmark and the 3DES application. Obviously, the more complex 3DES application consumes more power than the simple microbenchmark, where only one operation and two registers are involved.

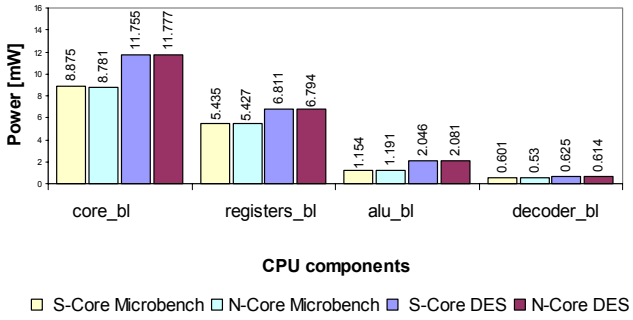

Fig. 6: Power comparison of normal and enhanced core for main CPU components

The energy that is needed by the two designs to perform the given tasks is shown in Fig. 7. In the case of the microbenchmark, the overall energy savings are about 20.8% compared to 8.7% for the 3DES application. This difference is due to the fact that the 3DES application has only one tenth of the cycles that are spent on the execution of the `xorldw` operation.

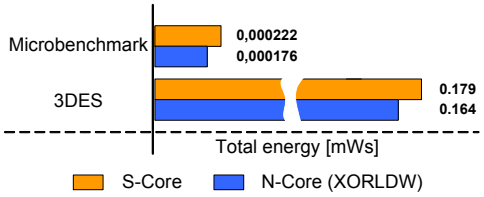

Fig. 7: Energy consumption of both designs

The object size of the 3DES routine that uses the super-instruction is 6% smaller (5572 bytes) than the object size of the normal instruction set (5924 bytes). This is a positive side effect for embedded systems such as our network processor, since embedded memory resources are often limited. 352 bytes of on-chip SRAM [7] occupy about 0.00924mm² in our technology. This is 100 times more area than the area needed for the super-instruction implementation. Please note that a typical SRAM with an appropriate size of 8 Kbytes (0.21mm²) that runs at the same clock speed as the CPU consumes 9.1mW of power and is therefore almost as important for the power estimation as the CPU itself.

5. Conclusion

We have shown the impact of instruction set extensions for general-purpose processors on power consumption and performance. Our approach enables us to extract promising super-instructions for application-specific domains. We have presented an instruction set extension that facilitates a noticeable speed-up and, at the same time, helps to save power. Due to its hierarchical structure, we have the opportunity to search a broad design space of potential instruction set extensions in a reasonable time.

Acknowledgements

The research described in this paper was funded by the Federal Ministry of Education and Research (Bundesministerium für Bildung und Forschung), registered there under 01M3062A. The authors of this publication are fully responsible for its contents. This work was supported in part by Infineon Technologies AG, especially the department CPR ST, Prof. Ramacher.

References

[1] M. Jain and M. Balakrishnan and A. Kumar. ASIP Design Methodologies : Survey and Issues. In Proceedings of the Fourteenth International Conference on VLSI Design, pages 76-81, January 2001.

[2] D. Langen, J.-C. Niemann, M. Porrmann, H. Kalte, and U. Rückert. Implementation of a RISC Processor Core for SoC Designs – FPGA Prototype vs. ASIC Implementation. In Proceedings of the IEEE-Workshop: Heterogeneous reconfigurable Systems on Chip (SoC), 2002

[3] J.-C. Niemann et al. A holistic methodology for network processor design. In Proceedings of the Workshop on High-Speed Local Networks held in conjunction with the 28th Annual IEEE Conference on Local Computer Networks (LCN2003), pages 583-592, 20 - 24 October 2003.

[4] M. Grünewald, J.-C. Niemann, and U. Rückert. A performance evaluation method for optimizing embedded applications. In Proceedings of the 3rd IEEE International Workshop on System-On-Chip for Real-Time Applications, pages 10-15, Calgary, Alberta, Canada, June 30 - July 2 2003.

[5] D. Fischer, J. Teich, M. Thies, and R. Weper. Efficient architecture/compiler co-exploration for ASIPs. In ACM SIG Proc. International Conference on Compilers, Architectures, and Synthesis for Embedded Systems (CASES 2002), pages 27-34, Grenoble, France 2002, Oct. 2002.

[6] C. W. Fraser, D. R. Hanson, and T. A. Proebsting. Engineering a simple, efficient code-generator generator. ACM Letters on Programming Languages and Systems, 1(3):213–226, Sept. 1992.

[7] Virtual Silicon. Single-Port SRAM Compiler UMC 0.13µm (L130E-HS-FSG), June 2003.