Feedback Driven Instruction-Set Extension

Uwe Kastens Dinh Khoi Le Adrian Slowik Michael Thies

{uwe, le, adrian, mthies}@uni-paderborn.de

Faculty of Computer Science, Electrical Engineering and Mathematics
University of Paderborn

ABSTRACT

Application specific instruction-set processors combine an efficient general purpose core with special purpose functionality that is tailored to a particular application domain. Since the extension of an instruction set and its utilization are non-trivial tasks, sophisticated tools have to provide guidance and support during design. Feedback driven optimization allows for the highest level of specialization, but calls for a simulator that is aware of the newly proposed instructions, a compiler that makes use of these instructions without manual intervention, and an application program that is representative for the targeted application domain.

In this paper we introduce an approach for the extension of instruction sets that is built around a concise yet powerful processor abstraction. The specification of a processor is well suited to automatically generate the important parts of a compiler backend and cycle-accurate simulator. A typical design cycle involves the execution of the representative application program, evaluation of performance statistics collected by the simulator, refinement of the processor specification guided by performance statistics, and update of the compiler and simulator according to the refined specification. We demonstrate the usefulness of our novel approach by example of an instruction set for symmetric ciphers.

Categories and Subject Descriptors

B.8 [**Performance and Reliability**]: Performance Analysis and Design Aids; C.4 [**Performance of Systems**]: Modeling Techniques, Measurement Techniques, Design Studies, Performance Measures.

General Terms

Design, Performance, Measurement, Experimentation.

Keywords

Instruction-Set Extensions, Compiler Generation, Simulator Generation, Encryption, Network Processor, Codesign.

LCTES'04, June 11–13, 2004, Washington, DC, USA.

1. INTRODUCTION

Special purpose processors are required to be cheap in cost, powerful in performance, and flexible in programmability. Even worse, the design of these processors has to meet aggressive time-to-market constraints. In the recent past, custom design ASICs have been a common response to these challenges. Meanwhile, application specific instruction-set processors (ASIPs) are becoming increasingly attractive since they allow to reuse efficient general purpose cores. Such a general purpose core can team up with a preferably small special purpose design covering the particular requirements of a specific application domain. Thereby, ASIPs intensify the need for tools that can detect application specific functionality, introduce corresponding instructions, and utilize these new instructions automatically.

The extension of instruction sets sketched above is typically driven by optimization goals that aim at speed-up, reduced code-size, reduced power consumption, and the like. Since the general purpose core we start with already experienced incremental improvements during previous design cycles, it does not seem feasible to accomplish further improvements without insight related to a specific application domain. And even if the application domain has been identified, application requirements and behavior may heavily depend on problem input size and application context, to mention a few. In order to achieve significant improvements in terms of the optimization goals stated above, it therefore seems mandatory to use feedback driven optimization. Thus we propose to evaluate envisioned instruction set extensions using cycle-accurate simulation and to make use of workloads that are representative for the targeted deployment scenario.

Since it is neither feasible to adapt the application software nor the compiler by hand to make use of newly introduced instructions, for example by explicitly calling intrinsic functions, a sophisticated toolchain has to conceal the additional complexity now encountered within the design phase as well as the deployment phase, such that the application software can readily be compiled for the resulting instruction set. Hence it is mandatory not only to propose an extended instruction set, but to have the optimizing compiler for that instruction set at hand, once the design phase is completed. In this paper we describe an approach to instruction set extension that, given an application domain with characteristic software, not only emits an optimized instruction set and a corresponding compiler, but that also provides an efficient cycle-accurate simulator. We would like to stress that we do not propose a fully automatic technique. Our inter-

est is to provide tools that empower a human designer to rapidly refine instruction sets on the basis of cycle accurate simulation. We expect our technique to significantly shorten the duration of a typical design cycle.

Within Section 2 we summarize our approach and contribution to the extension of instruction sets. In Section 3 we discuss previous work on that subject and point out notable differences. We outline in Section 4 how to generate optimizing compiler backends from concise processor specifications. The generation of corresponding simulators is given in Section 5 also makes use of these specifications. In Section 6 we detail our experimental setup and discuss results for the important application domain of symmetric ciphers.

2. WORKFLOW

Our goal is to tailor an existing processor to a specific application domain. We rely on an iterative process to refine the processor and the matching software development tools based on representative samples of domain-specific application software. Figure 1 provides an overview of this process. It starts with a formal model of the original 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 (see Section 4.3).

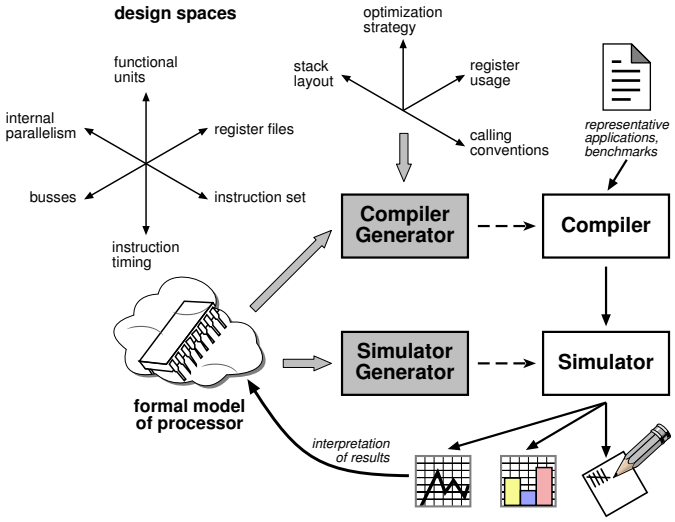

Figure 1: Iterative exploration of design space

Next, 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 easier 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 thoroughly 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, the number and capabilities of functional units, and the like. 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. As shown in [5] 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 [6]. A branch-and-bound algorithm traverses the relevant part of the design space and records all Pareto points without human intervention.

In this paper we are concerned with extensions to the instruction set of the processor exclusively. Additional instructions combine the effects of multiple pre-existing instructions, but execute faster. An exhaustive search through all pairs, triples, quadruples of instructions takes too long to be practical. Instead, our approach relies on a human expert in the area of hardware design to guide the process, that is to select feasible combinations of instructions. Our simulator determines the most prominent candidates for new instructions which would offer the highest gains in performance. A specialized visualization tool ranks and proposes these candidates to the expert.

After the constituents of new combined instructions have been selected, further software tools help to modify the formal model of the processor accordingly. This step requires some further input from the human expert; for example an estimate of the combined execution times. Due to the extensive tool support, performance evaluation based on the extended instruction set can commence within a couple of minutes. Iterative refinement of the instruction set stops, when the available opcode space has been exhausted, or when the diminishing performance gains of further combined instructions fall below a certain threshold.

3. RELATED WORK

Since the extension of instruction sets represents an active research area, we focus on notable differences to similar activities described in the literature. First of all is worth mentioning that we do not design entire instruction sets from scratch, but extend an initial instruction set that we adapt to a specific application domain similar to [2]. In this sense we pursue a different goal than work that aims at the design of optimal instruction sets for application specific instruction-set processors [9], or the design of ad-hoc architectures [3]. According to the classification used in [8], our approach is to introduce *fused operations*.

A number of publications propose to statically analyze data flow graphs to uncover opportunities for new instructions [10], [19], [1]. We use statistics provided by a cycle-accurate simulator running a characteristic application to guide that phase. Thereby, we take execution frequency into account and capture effects of loop control or conditionals. This even allows for fine-tuning according to very specific application needs, and is more in line with approaches described in [14], [13], that initially use profiling. In accordance with observations reported in [10], we focus on instruction pairs only.

Common to most approaches published recently is to use less precise cost functions to capture the impact of a new instruction. Since we iteratively introduce new instructions driven by priority indicated by execution frequency, then regenerate the compiler, recompile and rerun the application,

we also capture effects that relate to scheduling, register assignment, code size and similar effects which typically interfere in a complex manner. Thus we seamlessly capture reduced spill code, smaller instruction cache footprints, and the like. Also noteworthy, in our case it is the compiler's burden to make use of new instructions. In contrast to [13], [19], there is no need to recode the application.

Work described in [1], [13], [14], commonly builds on top of more or less complex graph cuts to represent candidates for new instructions. These cuts are constrained to yield multiple-input, single-out graphs (MISO) [13], [14], an extension to multiple outputs and convex cuts [1], or more complex user constrained cuts [1]. Since the number of cuts can grow exponentially in the number of nodes, these approaches take additional measures to deal with the combinatorial complexity. Typically, the search space is pruned using a top-N metric in combination with some priority ranking, for example execution frequency [14]. Our compiler's code selection uses tree patterns, which naturally translate into according MISO cuts. During instruction set extension, we combine complementary tree patterns by overlaying leaf nodes and root nodes, which we know to produce efficient results, although we enlarge tree patterns in a restricted way.

Work presented in [19], [8] also relies on a specification language (TIE, Tensilica Instruction Extension) to describe resource utilization and the like of new instructions. TIE is used to generate an efficient hardware implementation and a suite of software tools, including a compiler. At least in [19] TIE is not used to generate code selection, even though register allocation and scheduling happen automatically. New instructions have to be utilized by calls to intrinsic functions instead. In [8] the authors state that the compiler automatically makes use of these instructions, but no technical detail is given with respect to code selection. In both publications *estimated* performance is used to guide decisions.

The case study [18] summarizes a compiler/architecture codesign methodology that builds on top of the LISA processor design platform and the CoSy compiler environment. The authors also propose to use a compiler backend and cycle accurate simulator, both generated from concise specifications. The methodology does not focus on instruction set extensions but on architectural modifications that for example introduce additional functional units, or enlarge register files. Thus the simulator does not collect statistics related to instruction pairs and very little is said about implications for code selection.

In the interpreter domain, superoperators [16], [4], for example reduce the number of stack accesses during expression evaluation and thereby reduce interpretation time. We share the BURS technique with the generator *hit* [16], which emits assembly code for programs to be interpreted. However, it introduces new superoperators according to a metric that counts static occurrences of intermediate language constructs generated by the lcc ANSI-C frontend. These superoperators extend the lcc intermediate language. Assembly code for superoperators has to be provided and no specification language is involved. In [4], superoperators are determined during peephole optimization. The approach allows only for limited improvements, because the optimizer just considers adjacent instructions of the generated interpreter. We conclude that both techniques are suited to speedup the simulator we generate from incrementally refined specifications and hence they complement our technique.

4. INSTRUCTION-SET SPECIFICATION

Within this section we illustrate the set of specifications that describes the target processor for our development tools. These compact, high-level specifications enable us to generate a matching ANSI-C compiler automatically. We are mostly concerned with those parts of the specifications that capture the instruction set of the target processor and control the code selection phase of the generated compiler.

4.1 Original Processor

The unmodified Motorola S-Core CPU design (formerly M-Core [12, 11]) serves as the starting point for our extensions. It constitutes a rather small and simple RISC core which favors low power consumption over peak performance. The S-Core is a 32-bit integer-only architecture with 16-bit instructions for higher code density. 11% of that limited opcode space remains available for extensions to the instruction set. This makes the S-Core a viable foundation for a processor to be tailored to the networking domain.

While the pipeline structure of the S-Core is simple and uniform for the vast majority of instructions, the S-Core is not exactly an easy target for the compilation of (integer) C code. Due to the short instructions the S-Core offers only 16 directly accessible registers and restricts arithmetic operations to 2-address form. This requires conscious usage of registers in the generated code to avoid excessive data transfer between registers or spilling registers to memory.

Tight restrictions on immediate operands in many instructions demand distinct translation strategies in the compiler. For example, functions with stack frames larger than 32 bytes may reserve additional registers as auxiliary frame pointers for efficient translation. In contrast, data intensive functions with small stack frames need as many freely allocatable registers as possible. Compilation for the S-Core RISC processor is far from straightforward and competitive code quality requires state-of-the-art compiler techniques.

4.2 Code Selection

The code selection phase of our compiler employs bottom-up rewriting systems (BURS) [15]. Its specification takes the form of a tree grammar, where rules map fragments of intermediate language (IL) trees to sequences of target code. Each rule is annotated with costs which model execution time of the attached target code (see Figure 2). For each IL tree emitted by the compiler frontend, the code selector derives a tree cover with minimum total costs. This well-established, expressive approach can still be implemented very efficiently.

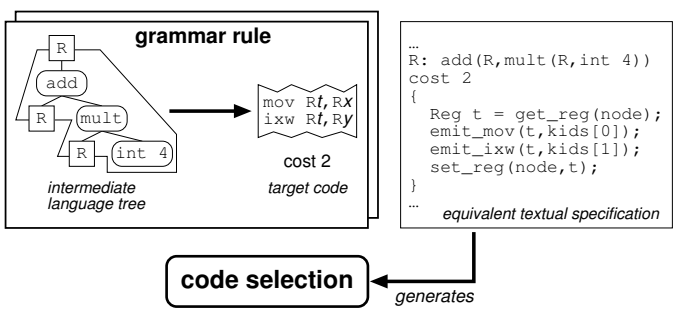

Figure 2: Specification of code selection

Figure 2 juxtaposes a grammar rule for the `ixw` instruction of the S-Core with the corresponding part of the actual processor specification. The `ixw` instruction supports indexed accesses to arrays with word-sized elements: `ixw` scales and adds the index in one step. To map the semantics of the IL tree onto the 2-address instructions of the S-Core, a temporary register and an additional `mov` instruction are needed. Register allocation eliminates this overhead, where permissible. The textual specification attaches actions to the grammar rule. Actions are specified as a fragment of C code that is invoked, when its rule has been applied as part of an optimal tree cover. Actions emit the individual instructions of the target code and manage any temporary registers required. Within the actions 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 subtrees below the current tree fragment.

We use an augmented version of the generator `burg` [7] to turn a tree grammar into the implementation of a tree automaton that performs code selection. `burg` conducts all cost computations at compiler generation time and encodes cost information in the states of the tree automaton. This leads to a very fast code selector which takes only constant time per IL tree node. Generation of the code selector is more expensive in terms of time and memory, but still fast: it takes less than 1 second wall clock time on today's desktop hardware.

Full cost analysis during generation of the code selector helps in designing and debugging larger tree grammars for complex processors (200–500 rules for RISC processors like S-Core, SPARC, PowerPC). The generator warns about incomplete rules sets, i. e., tree grammars that cannot cover every IL tree. At the same time superfluous rules are detected. These rules will never be used in cost optimal tree covers, because a cheaper alternative will always exist.

In this way, tree grammars can easily be developed incrementally. One starts with a minimal set of rules that cover one node of an IL tree each. Such a set of universally applicable rules ensures that valid target code can be generated for every input tree. To improve code quality, rules are added that exploit specialized instructions of the target processor, like the `ixw` instruction in Figure 2. Typically, these rules cover multiple nodes of an IL tree and emit a single equivalent target instruction.

The non-terminals of the tree grammar express how data is passed between sequences of target code that have been generated by different grammar rules. As the code selector combines only tree fragments with matching non-terminals, the associated code sequences are guaranteed to collaborate correctly after concatenation. Similar to the set of grammar rules, the set of non-terminals can be derived incrementally as well. Initially, a non-terminal is introduced for each elementary addressing mode of the target processor, e. g., access to a register or a constant (immediate) operand. Later, non-terminals can be added for address computations that are supported in load- and store-instructions and for the various ranges of values that may appear as immediate operands in different classes of instructions.

In our scenario, we extend this incremental development approach to the extended instructions that we add beyond the processor's original instruction set. Figure 3 depicts, how a new instruction (`xorlsr`) can be specified for the code selection phase. `xorlsr` applies a bitwise exclusive or to the result of a logical shift right by a constant amount. Part (a) of Figure 3 demonstrates the most direct approach to extend the tree grammar for `xorlsr`. A new grammar rule is added that combines the existing rules $R_{\texttt{xor}}, R_{\texttt{lsr}}$ for the constituent instructions `xor` and `lsr`. The target code of the new rule consists solely of the `xorlsr` instruction and its associated costs reflect the expected execution time, which is less than the total execution time of all constituents. If there are multiple rules that emit the same constituent instruction, the human designer is asked to pick a rule for the transformation of the tree grammar.

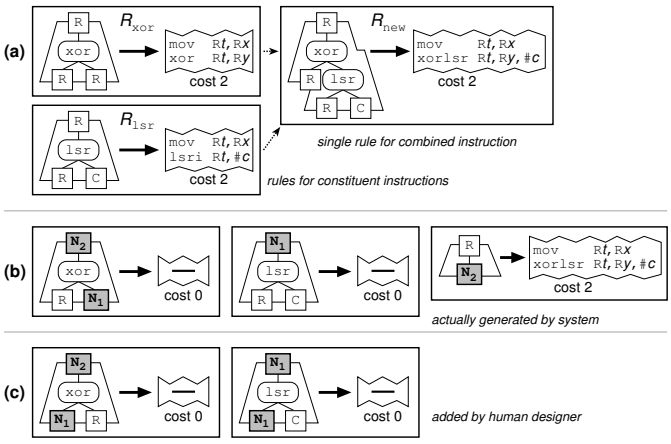

Figure 3: Specification of combined instructions

However, our system derives the new grammar rules in a different way, as shown in part (b) of Figure 3. First, new non-terminals N_1, N_2 are added to the tree grammar. Then, the system creates derivations of rules $R_{\texttt{xor}}, R_{\texttt{lsr}}$ that are linked by these new non-terminals. Note that these rules emit no target code. Code generation is deferred to an additional chain rule that emits the `xorlsr` instruction and thus converts the pseudo addressing mode N_2 to a conventional value stored in a register `R`.

The use of new non-terminals ensures that the `xorlsr` instruction is utilized for suitable expression trees only, but allows for easy extensions by the human designer. For example, the operands of `xor` are commutable and sequences of `lsr` instructions can be folded into one. These properties are exploited by the additional grammar rules in part (c) of Figure 3. They extend the scope of the new `xorlsr` instruction without a combinatorial explosion in the number of grammar rules.

4.3 Processor Abstraction

To automatically generate a compiler for a given target processor, an abstract model of the processor is needed. This model specifies those aspects of the processor that impact central data types and phases of the compiler's backend: representation of target code, code selection, scheduling, and register allocation. In addition, the model supports generation of a cycle-accurate simulator (see Section 5). While the model must capture all relevant aspect accurately, actual hardware design of the processor is of little concern. As a consequence, our processor abstraction is considerably smaller, simpler, and quicker to author or modify. These advantages carry over to exploration of the design space at our level of abstraction.

We use our own specification language UPSLA (unified processor specification language) to describe the target processor. UPSLA is a declarative language that relies on object-oriented techniques and algebraic concepts to obtain concise, manageable specifications which can be easily adapted to changes in the processor architecture. At the language level, declarations of abstract entities, inheritance among declarations, and equivalence classes of entities based on their behavior avoid redundancy and facilitate consistent global changes to the specification. UPSLA has been used to derive compilers for RISC processors of varying complexity. This includes architectures where effects of internal pipelining are visible to the programmer, as well as super-scalar processors, like the PowerPC 604 [17].

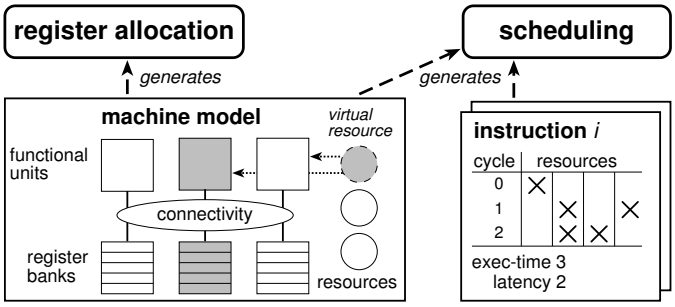

Figure 4: Components of processor model

Figure 4 shows the fundamental building blocks of a processor model in UPSLA. The model describes the available functional units and register banks of the processor, as well as their capabilities and their connectivity. Each instruction i is characterized by its execution time, its latency, and a plan of its resource usage during each cycle of its execution. The underlying set of processor resources constitutes a vastly simplified view of the actual processor design. Hardware resources that are used uniformly by all instructions and resources that never form a bottleneck can be left out. Complex interactions between multiple interrelated hardware resources can be expressed as a single *virtual resource* [17]. For example, the virtual resource in Figure 4 models a mutual exclusion between two functional units: only one of the attached units can acquire the virtual resource simultaneously. The virtual resource omits the detailed reasons, why this behavior of the processor emerges, e. g., restrictions in the hardware design to conserve chip area. Instead, the virtual resource simply enforces the observable constraints of the processor with the simplest means possible.

Due to its modest hardware complexity, the S-Core processor has been straightforward to specify in UPSLA. Nevertheless, the expressive power and flexibility of UPSLA has already been used to build a compiler that targets a small cluster of S-Core processors and conducts automatic fine-grained parallelization. However, here we are concerned with the instruction set of the processor exclusively. All other areas of the processor specification remain invariant during exploration.

To allow for easy addition of new instructions, our model of the S-Core covers all gaps in the opcode space with generic placeholder instructions. The operand encoding in these placeholders is extrapolated from the existing instruction decoder. Each additional instruction replaces a placeholder

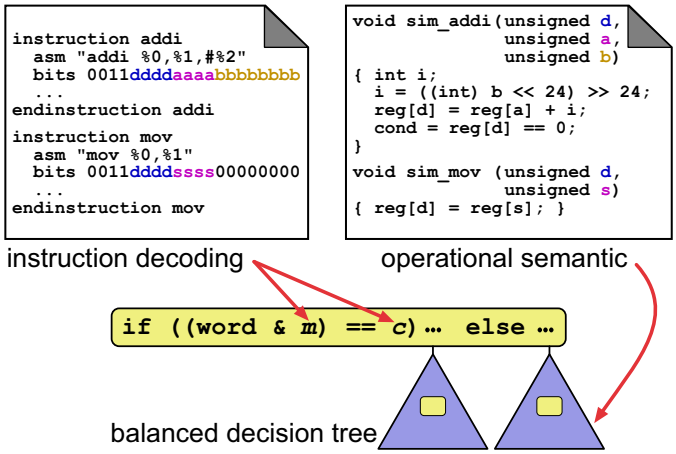

Figure 5: Generation of instruction decoder

with a suitable number of operands. The human designer simply adapts the declaration of the placeholder to match the properties of the new instruction. Usually just execution time and latency have to be modified. If the number of operands or their types differ between the two instructions, the declared resource usage of the placeholder needs to be changed as well. Hence, most exploration steps trigger only very small changes to the UPSLA specification. In addition, this approach prefers modifications that are easier to transfer to the actual hardware design later on.

5. SIMULATOR GENERATION

In our approach, the exploration of the design space must be fast and simple, because it is done repeatedly. Architectural designs, modifications thereof, and draft decisions affecting compiler construction must not call for too much manual intervention. Given a specific target architecture, we thus generate an appropriate compiler and corresponding simulator from a common specification, as shown in the figure 1. We then compile typical applications and suitable benchmarks and evaluate them on the target architecture. Therefore, the simulator picks up runtime information such as elapsed clock cycles, utilization of individual functional units, and frequency of instructions or instruction pairs, just to mention a few categories.

Central components of the simulator are generated from the specification of the target architecture that is also used for compiler generation. Since the simulator processes memory image(s), the simulator-generator needs information about the binary representation of individual instructions and their operational semantics. Given this input, it constructs a decoder that is tailored to the particular instruction set. A general hand-written framework interprets the recognized instructions and logs a wealth of statistics during the execution of each instruction.

5.1 Decoding and semantics of instructions

Our generator produces a balanced decision tree to decode instructions efficiently. By means of nested mask and compare operations, it maps opcodes to corresponding implementations, as shown on the left-hand side in Figure 5. As a side effect, it also checks the specification for ambiguities and conflicts of instruction encodings.

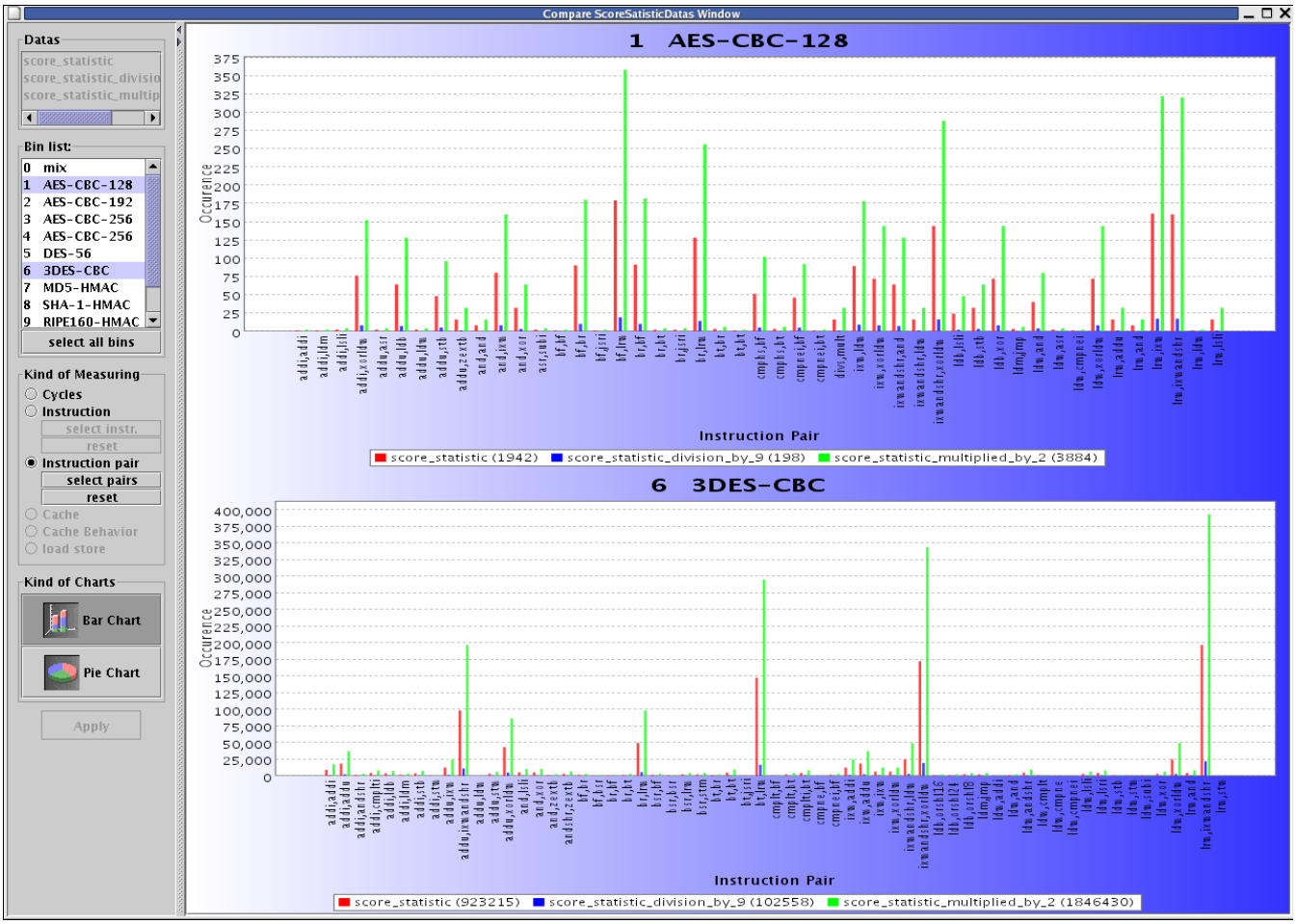

Figure 6: Performance visualization - occurrence figures for instruction pairs visualized by JScore

Operational semantics of the instructions are expressed in terms of ANSI-C fragments: For every instruction, the generator produces a C-function with a signature that adheres to the `bits` pattern described in the specification. One implements the function bodies for all instructions of the initial instruction set. As shown on the right hand side of Figure 5, the sample function `sim_mov()` has a signature with parameters `d` and `s`, according to the specification of the `bits` pattern of that instruction. The generator finally integrates these manually extended code fragments into a general hand-written interpreter framework.

If we add combined instructions to the original instruction set, the decoder must be able to decode and to interpret them without too much further ado. A combined instruction is introduced by an appropriate update of the UPSLA specification. The simulator-generator then produces a C-fragment which describes the effect of the combination. Therefore, it nests the existing implementations of the instructions that take part in the combination. The indications for the resulting latency as well as execution time given in clock cycles are also derived automatically, such that the new simulator can again gather cycle-accurate statistics.

5.2 Performance Statistics

To allow for a fast and simple evaluation of the target architecture by a human designer, we have implemented the performance visualization tool *JScore* shown in Figure 6. It takes data collected by the simulator and visualizes important characteristics of the executed program, e. g., elapsed clock cycles, execution frequency of instructions and instruction pairs, load/store bandwidth utilization, cache behavior and the like.

The simulator provides special functions which can be invoked to mark regions of interest within the application. The simulator picks up runtime information by region and organizes it accordingly, such that *JScore* can visualize separate regions or any aggregate thereof. This allows for simple hot-spot detection, since the designer can dissect the impact of arbitrary code fragments on the overall performance. In addition, the *JScore* permits it to accomplish certain operations such as multiplication with certain factors on the selected data records in order to simplify comparisons.

We would like to stress that instruction pairs found in the executed code not necessarily coincide with adjacent instructions, but expose direct data-flow dependences, i. e., they exchange data and hence are connected by def-use chains. This information is collected by the simulator and can be visualized by *JScore* as well. The emphasized instruction pairs that do not span beyond the (extended) basic blocks are the promising candidates for combined instructions.

6. EVALUATION

We illustrate the effect of our technique by example of a network processor with support for symmetric ciphers. We therefore compare the performance of a general purpose instruction set and an extended instruction set that has been customized to support the most important functions required by the IPSec protocol.

6.1 Application Domain

The application domain of network processors with support for cryptography is of increasing importance for general networking equipment and consumer communication devices alike. The most prominent and versatile protocol in use today for that purpose is IPSec, which provides authentication and encryption/decryption by means of different hash and crypto functions. Both types of functions are computationally demanding and typically dominate the workload of network processors. Moreover, IPSec is a security framework in the sense that additional functions can be plugged-in with little effort in order to extend the capabilities of an IPSec-device or to maintain standards conformance for evolving protocols. Thus performance and flexibility are both key for success within this application domain.

Currently, IPSec capable devices typically support the hash functions MD5-HMAC, SHA1-HMAC, and RIPEMD-160-HMAC, as well as the symmetric crypto-functions DES, 3DES, and AES. We make use of all these functions and utilize implementations taken from the OpenSSL library version 0.97a to evaluate our techniques for the extension of instruction sets. A simply structured driver calls these functions according to a custom workload profile (iMix - internet mix) which is commonly used for benchmarking in the networking domain.

6.2 Performance before ISE

The program we investigate throughout Section 6 invokes a couple of hash and crypto-functions on a synthetic packet stream (MD5, SHA1, and RIPEMD-160, DES, 3DES, and AES). The complete list is shown in Table 1 further below and is characteristic for VPN-devices shipped nowadays. The size of the packets adheres to the iMix specification. For the moment it suffices to observe that this benchmark takes 20.084.504 cycles to complete. This is the initial performance figure for the S-Core processor and the original instruction set. Including the very small driver, the code size of the entire program amounts to 110.144 bytes.

In addition to these figures, the optimizer has access to the occurrence counters depicted in Figure 7. It shows the top-30 statistics of instruction pairs that are candidates for combined instructions. From Figure 7 it becomes obvious that a few pairs dominate all other pairs; less than 10 pairs turn out to be significant in this particular case. The bar corresponding to the ordered tuple (`ldw`, `xor`) is marked with an asterisk and for example contributes more than 300k occurrences alone. Note that Figure 7 intentionally does not refer to the entire program, but to the program section that evaluates the performance of the combination 3DES, SHA-1. We will comment that decision soon. Obviously, the optimizer suggests to introduce a combined instruction `xorldw` to replace this particular instruction pair (`ldw`, `xor`).

6.3 Performance after ISE

The first combined instruction is introduced taking measures described in Section 4, which results in a refined processor specification. Thereafter, the compiler backend and simulator are regenerated, and the benchmark is run anew. Figure 8 below compares the resulting performance of 6 benchmark regions to that prior to optimization. A pair of bars belongs to one such region, the highlighted pair belongs to the program region we already used to format Figure 7.

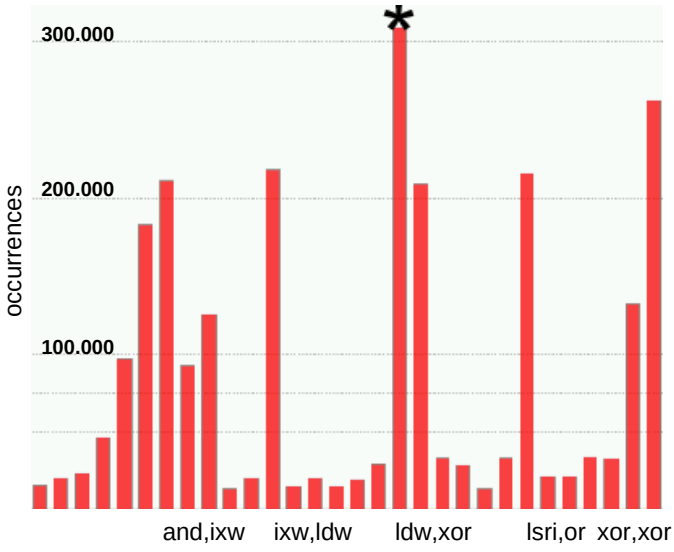

Figure 7: Initial top-30 instruction pairs

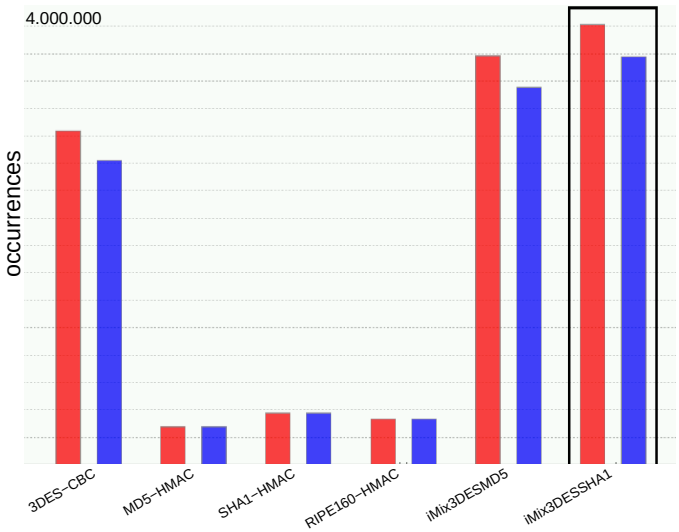

Figure 8: Impact of the combined instruction `ldwxor`

Figure 8 reveals that some regions benefit more than others. The region devoted to 3DES encryption for example takes 91.1% of the original cycle count to complete, while SHA-1 now still takes 100%. Table 1 indicates that the encryption functions benefit more than the hash functions do.

Figure 9 shows the top-30 instruction pairs with the first instruction extension applied. Bars relating to pairs that involve the new combined instruction `xorldw` are tagged with an asterisk. A total of 8 pairs is seen to utilize this new instruction either in its first or second component.

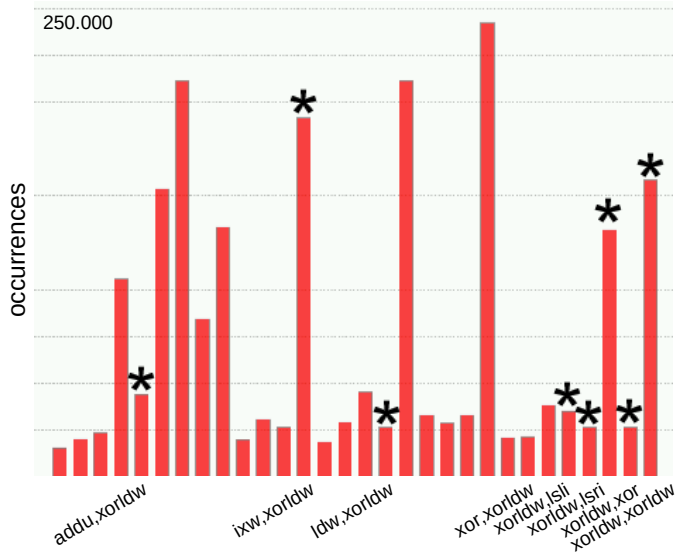

Figure 9: Top-30 instruction pairs including `xorldw`

Similarly, we introduce additional combined instructions. During the second iteration, the optimizer for example suggests to combine `and` (bitwise and) and `shr` (shift right) to yield `andshr`. Combinations chosen during subsequent iterations are shown in the according rows of Table 1. We applied a total of 5 optimizations. Few additional ones would have been applicable, but related occurrence counters did not meet the threshold for the reported experiment.

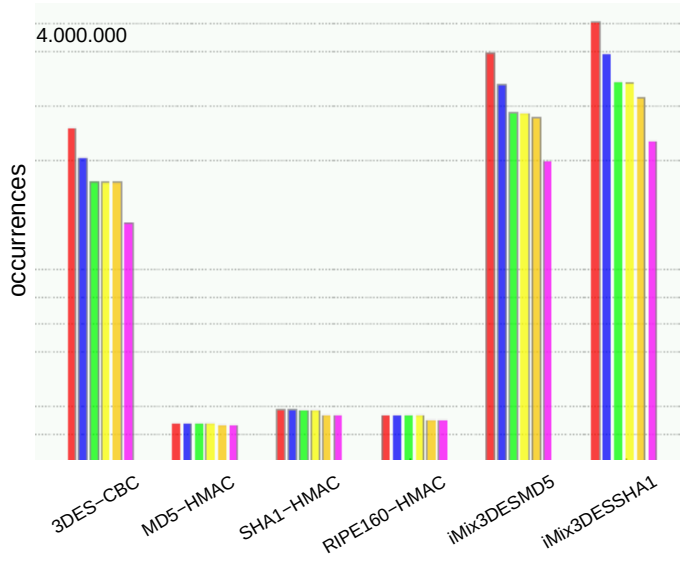

Figure 10: Performance after optimization

Figure 10 contrasts the performance improvements of 6 program regions by iteration of the optimization process (from left to right). Every region is seen to benefit from the combined instructions, refer to Table 1 for detailed figures.

Figure 11 reveals why no further significant improvement can be achieved. Combined instructions tend to team up, such that in Figure 11 the top-6 pairs involve at least one already combined instruction. Unfortunately, these pairs often take thus many arguments (registers, immediate operands, etc.) such that the resulting combined instructions do not fit into the instruction format of the S-Core processor. Adaption of the the instruction format is beyond the scope of this paper, but would allow for additional combined instructions. Note that the optimizer managed to introduce one such complex instruction (`ixwandshr`) in the 5th iteration.

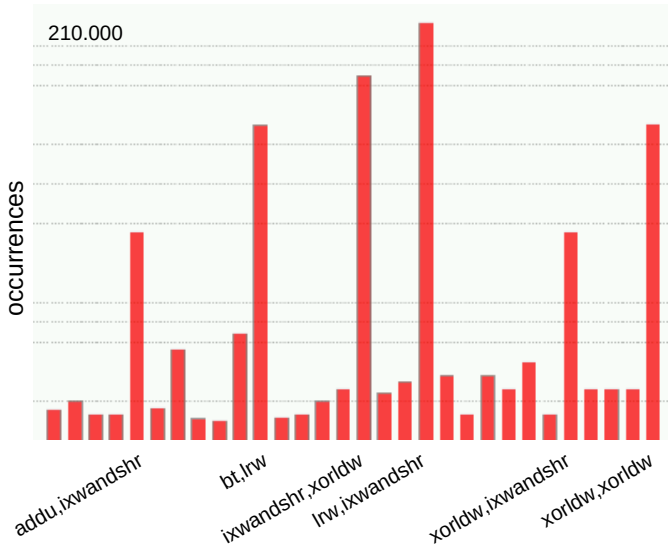

Figure 11: Instruction pairs after optimization

Table 1 compares the performance of different program regions (columns 2 through 11) for all suggested combined instructions, while the column devoted to code size (column 12) reflects the size of the entire executable program.

6.4 Performance of the simulator

We have conducted our experiments with the generated simulator on a Intel(R) Pentium(R) 4 CPU 2,4 GHz platform. According to the measurements of simulation performance, the S-Core processor would achieve 7,63 MHz for the IPSec benchmark described above. Thus an Intel(R) Pentium(R) 4 CPU with 2,4GHz needs approximately 313 clock cycles to simulate one clock cycle of the S-Core processor. It is worth mentioning that this is not the best performance our simulator can achieve, because it also has to collect a wealth of statistics. Disabling performance statistics provides a significant performance boost.

In order to verify the accuracy of our simulator we compared the emitted performance figures with those of an accurate VHDL-simulator. The simulation on the VHDL-level took substantially longer than our simulation (about 3 orders of magnitude), but the results in terms of clock cycles corresponded with figures emitted by our simulation.

6.5 Further remarks

During the optimization of an instruction set, it may happen that a combined instruction simply maps to a single instruction of the initial instruction set. The openssl source code for example relies on a preprocessor macro that mimics bitwise rotation of 32-bit operands. This macro typically compiles into a large expression tree in terms of the interme-

combined instruction	encryption & decryption					authentication			IPSec		code size
	AES-128	AES-192	AES-256	3DES	DES	MD5	SHA1	RIPE	3DES MD5	3DES SHA1	
xorldw	100.0%	100.0%	100.0%	91.1%	91.1%	100.0%	100.0%	100.0%	92.2%	92.8%	99.7%
andshr	93.1%	92.6%	92.1%	83.8%	83.9%	98.5%	99.0%	99.9%	85.4%	86.4%	99.1%
lsor	93.1%	92.6%	92.1%	83.7%	83.9%	98.0%	99.0%	99.6%	85.1%	86.3%	97.8%
rotli	94.6%	94.1%	93.7%	83.7%	83.9%	94.0%	89.1%	89.9%	84.1%	82.8%	87.9%
ixwandshr	**88.8%**	**87.9%**	**87.3%**	**71.5%**	**71.8%**	**94.0%**	**89.1%**	**89.9%**	**73.4%**	**72.9%**	**87.2%**

Table 1: Impact of combined instructions on performance and code size

diate language, such that the compiler backend specification may lack an appropriate pattern for code selection. During our experiments, such a pattern emerged automatically (called `rotli` in Table 1) and the code selector rediscovered that important code pattern. This optimization alone reduced the code size by 10% and contributed a speedup of also 10% (SHA-1).

The introduction of a combined instruction can cause the occurrence counters of instructions that do not participate in the new instruction to increase or decrease significantly, even though one might expect the effect of the combined instruction to be local in scope. As a consequence, the top-N instruction pair statistics may look completely different after one optimization. This implies that the recompiled program has to be profiled anew to guide further optimizations precisely. This is in line with our approach to regenerate the compiler backend, simulator and to profile the application anew for every modification of the instruction set.

7. CONCLUSION

We have presented a technique that allows to adapt a general purpose instruction set to a specific application domain at negligible cost in terms of time and manual intervention. This iterative technique is not meant to be fully automatic. Instead, we consider it to be an important complementary tool to be used by a human designer.

Our technique relies on a concise processor specification to generate a compiler backend and cycle-accurate simulator, and a small benchmark program that is characteristic for the application domain. By example of crypto-functions taken from the IPSec-protocol, we showed that our technique reduced the execution time of characteristic IPSec configurations by up to 28.5%. Moreover, the experiments showed that even though the optimization was guided by performance figures of a particular program region (3DES+SHA-1), the chosen optimizations also payed-off for other program regions. The AES-encryption, MD5 hash function, and other IPSec configurations also took significant benefit, which is not a bad result for an optimization process that hardly exceeded one hour of manual work.

We expect our technique to achieve similar improvements for other application domains as well. Current work investigates the benefit of instruction-set extensions for entire networking devices like DSL access multiplexers, QoS-capable routers, and storage-area devices.

8. ACKNOWLEDGEMENT

This work was conducted in context of the GigaNetIC Project supported by the German BMBF.

9. REFERENCES

[1] K. Atasu, L. Pozzi, and P. Ienne. Automatic application-specific instruction-set extensions under microarchitectural constraints. In *Proceedings of the 40th conference on Design automation*, pages 256–261. ACM Press, 2003.

[2] H. Choi, I.-C. Park, S. H. Hwang, and C.-M. Kyung. Synthesis of application specific instructions for embedded DSP software. In *International Conference on Computer Aided Design (ICCAD-98)*, pages 665–671, N. Y., Nov. 8–12 1998. ACM Press.

[3] H. Corporaal. *Microprocessor Architectures – from VLIW to TTA*. John Wiley & Sons, 1998.

[4] M. A. Ertl, D. Gregg, A. Krall, and B. Paysan. Vmgen — a generator of efficient virtual machine interpreters. *Software Practice and Experience*, 32(3):265–294, Mar. 2002.

[5] D. Fischer, U. Kastens, J. Teich, M. Thies, and R. Weper. Design space characterization for architecture/compiler co-exploration. In *ACM SIG Proc. International Conference on Compilers, Architectures, and Synthesis for Embedded Systems (CASES 2001)*, pages 108–115, Georgia, Atlanta, USA, Nov. 2001.

[6] 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.

[7] 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.

[8] D. Goodwin and D. Petkov. Automatic generation of application specific processors. In *Proceedings of the international conference on Compilers, architectures and synthesis for embedded systems*, pages 137–147. ACM Press, 2003.

[9] A. Hoffmann, O. Schliebusch, A. Nohl, G. Braun, O. Wahlen, and H. Meyr. A methodology for the design of application specific instruction set processors (ASIP) using the machine description language LISA. In *2001 IEEE/ACM International Conference on Computer-Aided Design (ICCAD '01)*, pages 625–630, Washington - Brussels - Tokyo, Nov. 2001. IEEE.

[10] R. Kastner, S. Ogrenci-Memik, E. Bozorgzadeh, and M. Sarrafzadeh. Instruction generation for hybrid reconfigurable systems. In *2001 IEEE/ACM*

International Conference on Computer-Aided Design (ICCAD '01), pages 127–131, Washington - Brussels - Tokyo, Nov. 2001. IEEE.

[11] Motorola. *M-Core Reference Manual*, 1998.

[12] Motorola. *MMC2001 Reference Manual*, 1998.

[13] A. Peymandoust, L. Pozzi, P. Ienne, and G. D. Micheli. Automatic instruction set extension and utilization for embedded processors. In *Proceedings of the IEEE 14th International Conference on Application-specific Systems, Architectures and Processors, The Hague, The Netherlands*, June 2003.

[14] L. Pozzi, M. Vuletic, and P. Ienne. Automatic topology-based identification of instruction-set extensions for embedded processors. Technical report, Swiss Federal Institute of Technology Lausanne, Processor Architecture Laboratory, 2001.

[15] T. A. Proebsting. Burs automata generation. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 17(3):461–486, 1995.

[16] T. A. Proebsting. Optimizing an ANSI C interpreter with superoperators. In *Principles of Programming Languages (POPL '95)*, pages 322–332, 1995.

[17] E. Stümpel, M. Thies, and U. Kastens. VLIW compilation techniques for superscalar architectures.

[18] O. Wahlen, T. Gloekler, A. Nohl, A. Hoffmann, R. Leupers, and H. Meyr. Application specific compiler/architecture codesign: a case study. In *Proceedings of the joint conference on Languages, compilers and tools for embedded systems*, pages 185–193. ACM Press, 2002.

[19] A. Wang, C. Rowen, D. Maydan, and E. Killian. Hardware/Software instruction set configurability for System-on-Chip processors. In *Proceedings of the 2001 Design Automation Conference (DAC-01)*, pages 184–188, New York, June 18–22 2001. ACM Press.