Automated Instruction-Set Extension of Embedded Processors with Application to MPEG-4 Video Encoding

Nikolaos Kavvadias and Spiridon Nikolaidis
Section of Electronics and Computers, Department of Physics,
Aristotle University of Thessaloniki, 54124 Thessaloniki, Greece
nkavv@physics.auth.gr

Abstract

A recent approach to platform-based design involves the use of extensible/configurable processors, which are programmable architectures offering the possibility to customize the instruction set and/or underlying microarchitecture. Part of the designer responsibilities is the domain-specific extension of the baseline processor to fit the customer requirements. Important issues that determine the success of this process are the automated application analysis and candidate instruction identification/selection for implementation as application-specific functional units (AFUs). In this paper, a design approach that encapsulates automated workload characterization and instruction generation is utilized for extending processors to efficiently support video encoding kernels, as MPEG-4 shape encoding and block-matching motion estimation algorithms. The method used for instruction generation is a highly parameterized adaptation of the MaxMISO technique, which allows for fast design space exploration. It is proven that only a small number of AFUs are needed in order to support the entire range of examined algorithms and that it is possible to achieve 2× to 3.5× performance improvements although further possibilities such as subword parallelization are not currently regarded.

1. Introduction

Embedded processors for consumer applications, present interesting architectural refinements, in order to support performance-critical algorithms e.g. for video encoding with a favorable efficiency/flexibility tradeoff. In the development of such processors, closely matched design of instruction sets and micro-architectures is required, in respect to the application benchmarks, while being subject to several constraints. These constraints stem out of diverse and often conflicting requirements met in recent application areas for programmable SoCs such as low power consumption, performance in a given application domain, code size and overall system cost [1].

The challenge of delivering the optimum balance between efficiency and flexibility can be met with the utilization of customizable processors. Most commercial offerings fall in the category of configurable and extensible processors [2],[3]. Configurability lies in either setting the configuration record for the processor core (regarding different cache sizes, multiplier latency/throughput and technology specific module generation) [4] or allowing modifications on the microarchitecture template. Extensibility of a processor comes in modifying the instruction set architecture by adding single-, multi-cycle or pipelined versions of complex instructions. This may require the introduction of custom units to the execution stage of the processor pipeline and this should be accounted in the architecture template of the processor.

The designer freedom available in configuration/extension scenarios of customizable processors has to be exploited for advantageous domain-specific specialization. While it has been argued that complete application characterization is a demand for inhibiting mismatches between expected and delivered performance [5], the established approach follows a two-level strategy of a) focusing on aggressively optimizing the application kernels (e.g. inner loops mapped on VLIW templates through software pipelining) and at a subsequent phase examining its effect on the entire application. For this reason, MediaBenchII [6], a long-awaited update to the popular MediaBench benchmark suite [7], will incorporate kernels such as motion estimation (key procedure in video coding standards), and wavelet filters.

In this paper, an application analysis and custom instruction generation prototype framework is presented, based on the SUIF/MachSUIF compiler infrastructure [8] and a parameterizable instruction generation engine. Its features include support of a RISC-like instruction set with unallocated infinite resources and its close derivative backend, built-in area and delay early estimators for the AFUs, classic compiler optimization passes, arithmetic optimizations, and a highly-controllable version of the MaxMISO instruction generation algorithm [9] that enables interesting multi-dimensional design space exploration possibilities.

Our second contribution regards identifying common instruction-set extensions for popular video encoding kernels. By applying our approach on a media intensive stressmark suite focusing on motion estimation as the property of interest, we derive a minimal set of hardware extensions corresponding to only 3 application-specific functional units, which can support common functionalities across multiple profiles and algorithms.

The rest of this paper is organized as follows. Related work in application analysis, candidate instruction identification and especially instruction-set extensions for motion estimation algorithms is summarized in Section 2. The instruction generation approach for customizing embedded ASIPs is detailed in Section 3. Section 4 discusses the application of the proposed approach on an MPEG-4 compliant shape encoder as a case study, and it is further applied on a set of motion estimation algorithms. Finally, Section 6 summarizes the paper.

2. Related work

Methodical research efforts on application-specific extensions regard automating methods to explore the architecture design space [10],[11],[12],[13],[14]. A few instruction generation frameworks that can be directly evaluated for instruction-set extensions exist [10],[15], with open specifications and open-source instruction clustering tools of polynomial-time complexity [16]. An advantage of their approach is the proposal of a pattern file format usable among primitive and extension instructions for storing, manipulating and exchanging instruction patterns. Some issues with the Pattlib approach regard the significant efforts for adapting the GCC compiler to emit information in “pattlib” format, and that the intermediate representation (IR) for their selected backend (SPARC V8) is not architecture-neutral. A disciplined approach to custom instruction generation for extensible processors is found in [14] where the Xtensa processor is augmented with application-specific instructions that may combine VLIW, SIMD or fused (chained) RTL operations. This method is application profile-driven and it borrows exploration and bitwidth analysis features from the mature PICO system [17]. However, it is strongly oriented towards the Xtensa architecture template, applying strict restrictions on the design space since it considers only 2-input MISO single-cycle instruction candidates. In addition to that, although they have included MPEG-4 video

encoding as one of their benchmarks, detailed information on algorithms (e.g. for motion estimation) and their implementations is omitted.

Close to our aims but from an empirical viewpoint, recent ARM architectures (ARM9 and later) can be optionally augmented with a low-power programmable coprocessor named MOVE [18], for accelerating the SAD operation in motion estimation. The plain SAD calculation without regarding data transfers requires 2 or 3 cycles given the operating mode. However, due to the effect of the limited bandwidth for accessing the pixel values, a 2× speedup for MPEG-4 encoding over a software implementation has been reported by the developers, compared to a 8-fold theoretical speedup assuming transparent data updates for the block buffer and their use of SIMD. Also, the authors do not provide sufficient performance measurements over a range of motion estimation kernels running on a MOVE-enhanced system.

Regarding computational complexity measures and PSNR quality results for motion estimation algorithms, there exist a number of credible studies including [19]. However, in the vast majority of these works, low-level complexity metrics have not been thoroughly obtained. Usually, coarse-level measurements are collected as dynamic instruction mix for a specific target architecture (e.g. a Sun SPARC in [19]), and number of search positions in the reference window, since these efforts have primarily focused on: a) proving that fast motion estimation algorithms are competent to exhaustive search in terms of psychovisual quality and b) estimating the relative complexity against the reference full-search algorithm. More elaborate characterization of these algorithms for the purpose of instruction-set extension necessitates the early extraction of complex instruction candidates that would accelerate the application's execution in programmable environments.

3. Instruction generation procedure for customizing domain-specific embedded processors

It is often at early stages in processor design, that the compilers and simulators for the design space of applicable processor architectures are not available. As a common estimation platform, so that application characterization results are useful to the spectrum of evaluated microarchitectures, we use the MachSUIF IR, which represents a generic RISC, not biased towards any existing architecture. The application IR can be organized into control data flow graphs (CDFGs) of the procedures in the program. Dynamic characterization is performed on the host machine by executing the resulting C program, generated by translating the SUIFvm (SUIF virtual machine instruction set) back to three-address C code.

The instruction generation flow, which is shown in Fig. 1, uses an enhanced version of the application characterization environment discussed in [20]. On Step 1, the input C code for the application is processed by the SUIF frontend [21], which involves AST construction with SUIF nodes, and C-level statement dismantling transformations. On Step 2, the resulted representation is fed to the *s2m* pass to emit SUIFvm assembly-like IR. The IR code is unscheduled while complete procedure entry and exit sequences have not been inserted at this stage, since stack frame layout is highly processor dependent. It is not meaningful to seek useful instruction extensions in stack manipulation code since in this case false dependencies within the data flow graphs of each basic block are created [12]. Step 3 performs architecture-independent optimizations on the IR, such as a) peephole optimization, b) constant propagation, c) dead code elimination, d) early operator strength reduction and e) local common subexpression elimination (LCSE) [22] to optimize the SUIFvm assembly.

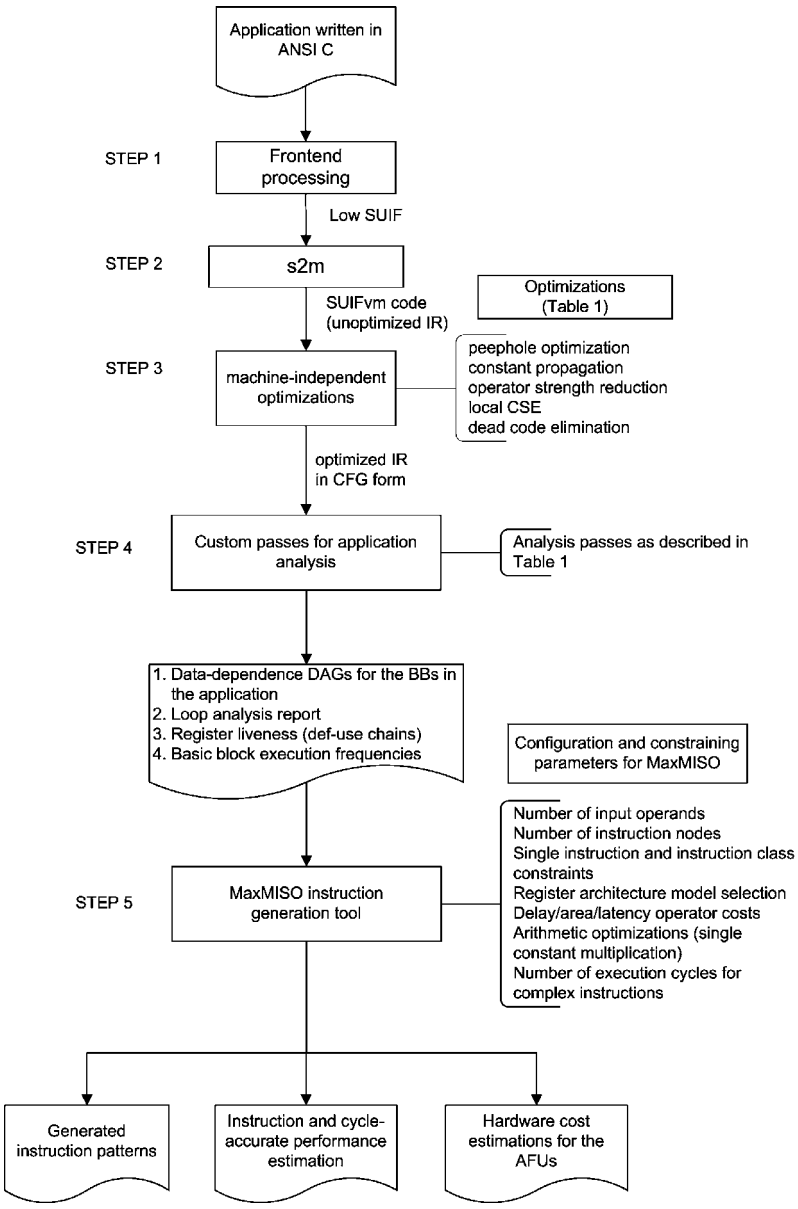

Figure 1. Application analysis and parameterised instruction generation flow.

On Step 4, specific static and dynamic profiling information for the input application is gathered with the help of a set of analysis passes, accepting SUIFvm IR in CFG form. Table 1 gives compact descriptions of analysis and transformation passes that have been added to MachSUIF.

The instruction generation process takes place on Step 5. The instruction identification and generation engine currently implements the MaxMISO (maximal multiple-input single-output) algorithm [9], which identifies the maximal non-overlapping connected subgraphs of the data-dependence directed-acyclic graph (DAG) that produce a single computation result. In its original form, the only applicable constraints regard the maximum number of input operands that can be delivered to the AFU, but in our implementation is enhanced in a number of ways to be more suitable for performance tradeoff analysis. These MaxMISO parameters include:

Table 1. Custom analysis and transformation passes for the MachSUIF compiler infrastructure.

Pass name	Description	Original source
dagconstruct	Construct data-dependence graphs for each basic block	In-house
instrmix	Generates static instruction mix	In-house
liveanalysis	Builds definition-use chains and applies liveness analysis using the MachSUIF *cfa* library	In-house
loopstr	Invokes the built-in natural loop analyzer [Aho86] of the MachSUIF *cfa* library	In-house
m2c_bb	GNU diffs and Perl scripts for correcting m2c output and generating an instrumented version of the 3-address C output of *m2c*	In-house
buildcg	Call-graph constructor	In-house
lcse	Applies local common subexpression optimization	[22]
if_conversion	Applies if-conversion optimization which can be used with instruction sets that have predication support	[22]
cplx_locate	Locates portions of SUIFvm code that can be replaced by calls to the built-in SUIFvm complex instructions: *abs*, *min*, *max*.	In-house
strength_reduct	Simple operator strength reduction for multiplication and division	In-house

1) The maximum number of primitive instruction nodes to be included in the MaxMISO.
2) The establishment of two types of node constraints that can be applied to any instruction class:
   a) Type-A or *boundary-node* constraint: Applying this constraint, prohibits growing an instruction cluster beyond the specified instruction. It has been observed that its application on data transfer instructions (load, store, memory copy), forces the generation of complex addressing modes. This procedure automates a traditionally ad-hoc portion of the ASIP design flow, which is the identification of the most beneficial addressing modes for the processor's data transfer instructions.
   b) Type-B or *node-inclusion* constraint: A constraint of this type will not permit the inclusion of the specified instruction in the MaxMISO under build. It is usually applied to control-transfer instructions (*cti*) such as conditional/unconditional branch and call/return operations. In the majority of extensible processors, the end-user is not permitted to alter the control transfer mechanisms i.e. to add complex instructions to the original ISA that modify the instruction fetch path, since its effect to the processor cycle time is less predictable than in the case that instruction extensions reside solely on the execution pipeline stage(s) of the processor.
3) Applying a limit on the maximum number of hardware cycles that is required for instruction execution. Single-cycle instructions require only minor modifications to the main instruction decode logic while multi-cycle instructions demand additional FSM control and possibly user-defined state registers [12],[23].

Additional features of our MaxMISO implementation include:

1) Support for SUIFrm (SUIF real machine), a close backend ISA to the SUIFvm IR that allows register allocation and code generation.
2) Single constant multiplication optimization according to Bernstein's algorithm [24],[25] but with exact delay costs instead of cycle costs.

3) Multi-operand addition optimizations [26],[27] that currently are applied after the main instruction generation procedure.

4. Motivational example: MPEG-4 context-based shape encoding

As demonstration application, we have selected a context-based shape encoder for MPEG-4 [28],[29]. This encoder allows for lossy decisions in the encoding of shape information, and is controlled by a motion vector related (*motion_th*) and a lossy compression (*alpha_th*) parameter. Parameter *motion_th* allows controlling the accuracy of motion vectors: *motion_th*=0 corresponds to maximum (pel) accuracy while a non-zero *motion_th* value means that for low-motion macroblocks, a motion vector prediction is encoded in the bitstream instead of running motion estimation for the given macroblock. The *alpha_th* block threshold (*alpha_th*) allows lossy compression: each block is set to all-0 or all-255 mode depending on its accepted quality compared to the threshold. In our experiments, the key frame of each P-VOP sequence (typically 10 frames) is intra-coded while the remaining frames are inter-coded.

We have obtained execution profiles of this application for the 16 run-cases defined by the following parameter space: {*alpha_th*, *motion_th*}={0, 32, 64, 256}. It was derived that the most performance critical basic blocks are BB #40 and #41 of the *find_vects* procedure, which is the actual code segment for SAD computation. Table 2 summarizes statistics for the generated custom instructions and execution cycles measures for the instruction-set extensions under 3 different constraint scenarios and for specific number of inputs ("#inputs" column). In columns "#MaxMISO" and "MaxMISO size", the number of static occurrences of custom instructions and their size in number of primitive instructions are given respectively. It is deduced that the constraint on the number of inputs has a dramatic impact on the size of the generated MaxMISO. The achievable speedups range from 1.07 to 3.50, the latter for the case with no restriction on the number of inputs. This case can only be realized with the use of state registers that significantly reduce the demand of input operands from the register file.

From this point forward, each basic block is assigned a unique name of the form: <*procedure_name*>.<*basic_block_number*>. Figure 2a shows the data-dependence graph of the maximal speedup MaxMISO identified in basic block *find_vects*.40 under node constraints: {Type-A, Type-B} = {*str*, *cal*}. Area and delay metrics in Fig. 2 are normalized to the values for a 32×32-bit single-cycle multiplier returning a 64-bit result (not truncated). By observation of the data flow graph in Figure 2a we can extract some important remarks:

1) There exist 3 different instances of multi-operand addition with 3-, 4-, and 5- input operators. Larger collapsing addition structures could be devised if the constant multiplications (discussed in the following remark) are incorporated into corresponding multi-operand adders but this feature has not yet been automated. We have characterized multi-operand adders from a public VHDL library [27] for our calculations. Figure 2b shows the MaxMISO with all occurrences of multi-operand addition substituted by optimized circuits based on carry-save *n*:2 compressors.
2) There is no variable-by-variable multiplication involved, but only multiplications-by-constant. This constant is the picture width (#352 for test sequence "stefan"). Typical picture widths for streaming video are: 120 (SQCIF), 176 (QCIF), 352 (CIF), and 704 (4CIF). The design in Fig. 2c further reduces both delay and area compared to Fig. 2b by utilizing single constant multipliers. In case a set of constants has to be supported, multiple constant multipliers should be used instead.

3) The 2 memory load operations (*lod*) do not correspond to zero-successor nodes of the DFG. Generally, load/store instructions infer a node-inclusion constraint, if the corresponding AFUs are not connected through parallel paths to the data memory.
4) The application of boundary-node constraints for memory access instructions, partitions the DAG of Fig. 2a into three non-overlapping regions at *lod* boundary. A theoretical speedup limit (without regarding delay costs for the primitive operators) of 4.3 is estimated for this basic block, which results in a 2.46× speedup for the entire algorithm.
5) A particularly low ILP (instruction-level parallelism) of only 1.6 is calculated for the DFG of Fig. 2a and an ILP of 2.18 has been obtained on a 4-way processor configuration simulated with SimpleScalar 3.0d [30] for the entire application. The compiler used is *gcc*-2.7.2.3 targeted to the SimpleScalar architecture with the –O3 optimization flag turned on. It can be argued that low ILP, privileges architectures that exploit chained against VLIW operations. In contrast to VLIW architectures issuing simple independent operations each one occupying an instruction slot, processors with chained instruction extensions, exploit dependent operations. For this reason, these architectures have potential of higher utilization against their VLIW counterparts with much less hardware resource demands, for the case of low ILP. Notably, media-centric applications such as MPEG-4 visual (version 1 and 2), and H.264/AVC have respectively low ILP of about 2 in average for both the encoding and decoding applications [6].

Table 2. Detailed statistics for the generated custom instructions, extracted from performance-critical basic blocks of the shape encoder.

Basic block	#inputs	Node constraints		#MaxMISO	MaxMISO size	Est. relative %cycles	Speedup
		Type A	Type B				
find_vects.40	2	str	cal	5	2	73.68	1.36
find_vects.41	2	str	cal	2	2		
find_vects.40	2	str, lod	cal	5	2	73.68	1.36
find_vects.41	2	str, lod	cal	2	2		
find_vects.40	2	str, lod	cti	5	2	86.61	1.15
find_vects.41	2	str, lod	cti	2	2		
find_vects.40	4	str	cal	6	2.33	47.37	2.11
find_vects.41	4	str	cal	2	3		
find_vects.40	4	str, lod	cal	6	2.67	47.37	2.11
find_vects.41	4	str, lod	cal	2	3		
find_vects.40	4	str, lod	cti	6	2.67	65.22	1.53
find_vects.41	4	str, lod	cti	2	2.5		
find_vects.40	∞	str	cal	1	16	28.57	3.50
find_vects.41	∞	str	cal	1	6		
find_vects.40	∞	str, lod	cal	3	4.67	43.61	2.29
find_vects.41	∞	str, lod	cal	2	3		
find_vects.40	∞	str, lod	cti	3	4.67	48.78	2.05
find_vects.41	∞	str, lod	cti	2	3		

The final AFU hardware (Fig. 2d) for the MaxMISO of Fig. 2a is additionally optimized, by the manual introduction of state registers, which is motivated by analysis of register liveness results. The maximum speedup can be approached with 8 state registers as can be seen in Fig. 2d. Also, selection of operator bitwidths has been performed with nodes *sub* and *abs* operating on 8-bit and the remainder nodes on 16-bit quantities. Overall, automated and manual optimizations result in 85% and 40% reduction against the unoptimized hardware for the derived MaxMISO for area and delay metrics, respectively.

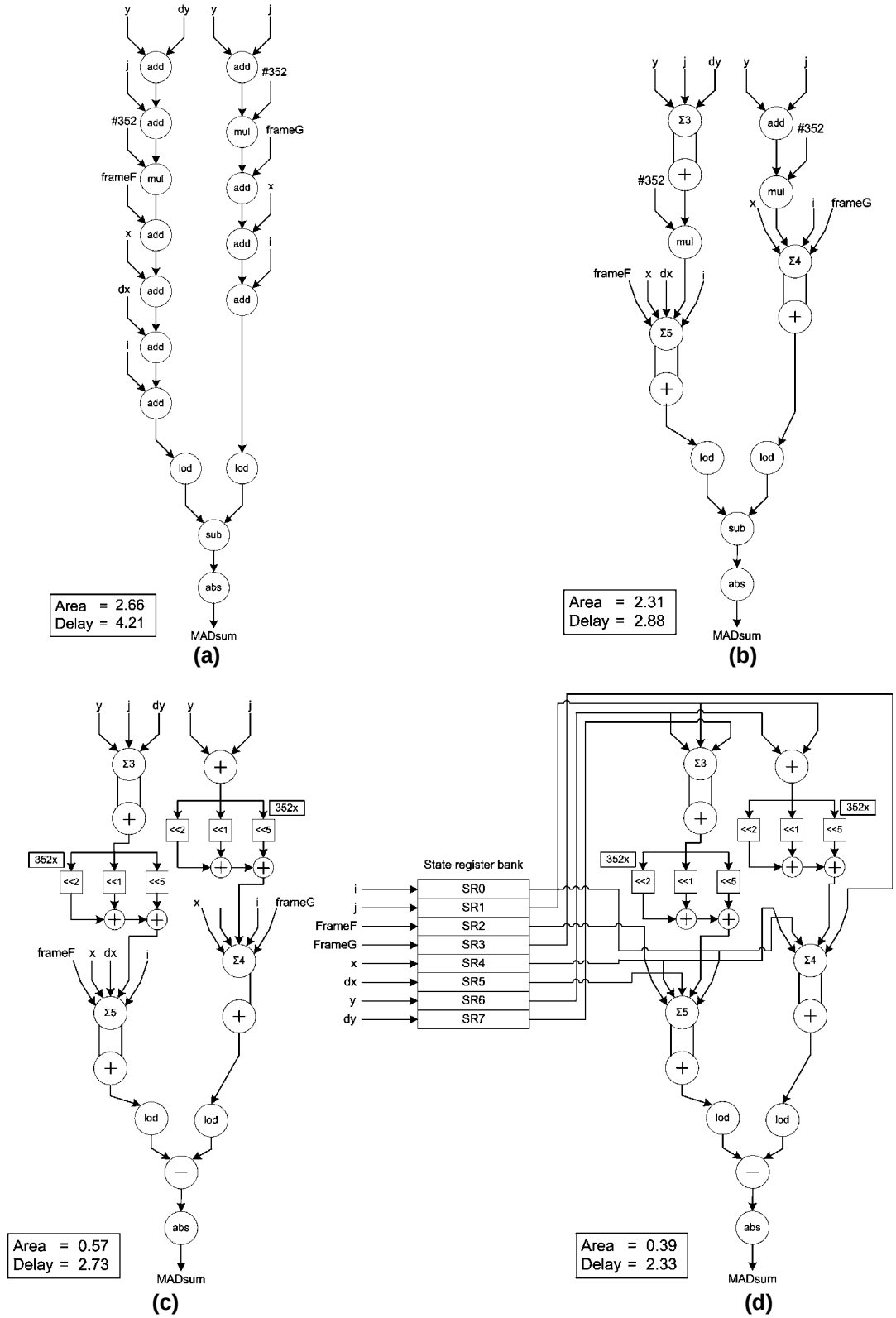

Figure 2. Optimization steps for the MaxMISO identified in find_vects.40. (a) Initial DFG. (b) Multi-operand addition instances in the DFG. (c) Utilization of constant multiplication. (d) Utilization of state registers.

5. Application analysis, design space tradeoffs and instruction extension generation for a motion estimation stressmark

At this point, we will evaluate the proposed approach for instruction-set extension of block-matching motion estimation algorithms, which are used in MPEG video compression systems. Table 3 presents the algorithmic kernels under investigation [31]. From left to right, the abbreviation, a short description an appropriate reference and the number of executed instructions are given.

Table 3. Summary of the motion estimation algorithms included in the stressmark suite.

Abbreviation	Description of motion estimation algorithm	Reference	Exec. instructions
fs	Full-search	--	524006501
ofs	New full search	??	85480757
3ss	Three-step search	[32]	66407224
4ss	Four-step search	[33]	55404110
n3ss	New three-step search	[34]	67789322
bbgds	Block-based gradient descent search	[35]	47494185
coher	Region-based optical flow motion estimation	[36]	58004136
liu	Full-search with 4:1 alternate pel subsampling	[37]	110970080
pfs	Partial distortion full search	[38]	156713876
ucbds	Unrestricted center-biased diamond search algorithm for motion estimation	[39]	53705554

5.1. Application analysis

Figure 3 presents the dynamic instruction mix results for the kernels of Table 3. Instructions are divided into integer and floating-point, while each of those has distinct subtypes: load and store, arithmetic, logical, shift, multiply, division, unconditional and conditional branch, and call/return and remaining instructions. It can be seen that arithmetic operations dominate the instruction mix with a 62% weight. For a processor with hardware support for *abs* instruction, the contribution of arithmetic instructions is about 72%.

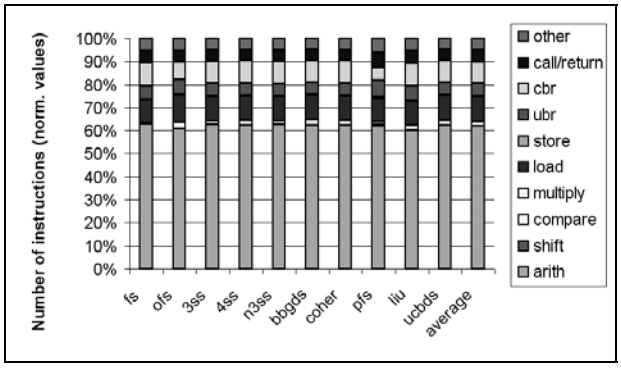

Figure 3. Dynamic instruction mix for the motion estimation stressmark.

5.2. Design space exploration and tradeoff analysis

As a first task in tradeoff analysis for the motion estimation stressmark, we identify the maximum performance increase that can be achieved by adding support for MaxMISO

instruction clusters to the base microarchitecture. In Figure 4, normalized execution cycles are plotted against the maximum number of inputs for given cycle constraints for the average case of all kernels shown in Figure 4b. This case resembles hypothetical reconfigurable hardware, which could support a large number of custom instructions, limited in principle by the configuration cache requirements, with single-cycle partial reconfiguration capability. This would be true for certain schedules that take the maximum utilization for the main processor datapath and the custom logic. We can see that for up to 2-input instructions, a 2× speedup is expected, while for 4-input instructions it increases to 2.5 and for the asymptotical case of unconstrained number of inputs, a theoretical maximum of 3.5 is observed. It should be noted, that subword parallelism capabilities and generally bitwidth optimizations have not been considered but a number of algorithms for bitwidth analysis [39],[40] are currently under study for extending our framework. Last, it appears that constraining the number of cycles per instruction does not affect performance in terms of execution cycles for a small to moderate number of inputs.

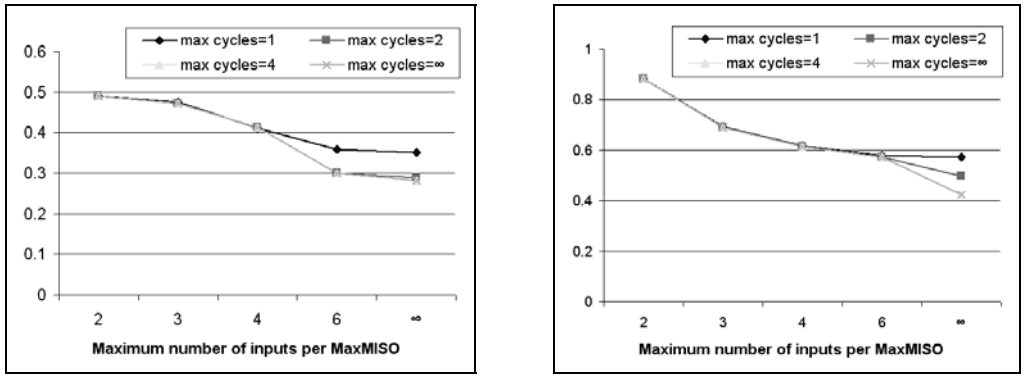

Figure 4. Theoretical maximum normalized execution cycles for: (a) the average case of motion estimation algorithms. (b) the MPEG-4 shape encoder.

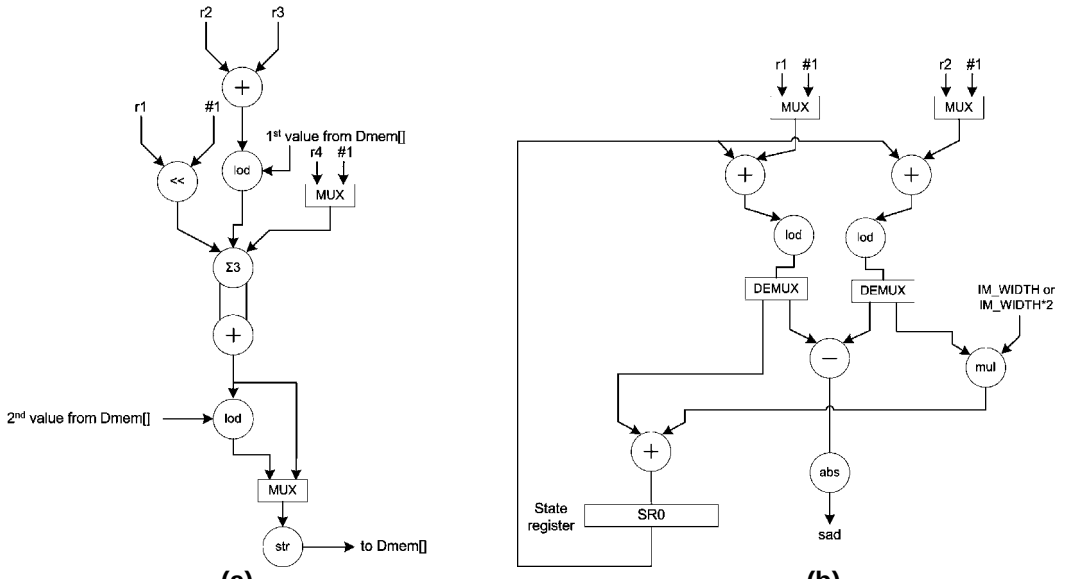

Figure 5. DFGs representing AFU hardware for common custom instructions among motion estimation algorithms. (a) DFG for the res_image.9 basic block. (b) View of a common single-cycle SAD implementation unit.

Further in analyzing the applications, we restrict our interest to the critical basic blocks, and to custom instructions of up to four input operands, that could be supported by configuring a dual register file. It has been derived that significant performance increase can be achieved with only three additional functional units for supporting the shape encoder and a variety of motion estimation algorithms. DFG views for the two AFUs extracted from the stressmark are given in Fig. 5, while the AFU for the custom instruction for shape encoding acceleration is shown in Fig. 2d. The common characteristic for the AFUs in Fig. 2d and 5a is the need to load two data values from the data memory, which provide the input pixel data for the SAD operation. We should note here that if data memory prefetching techniques are not used, at least 2 cycles are needed for the instructions in Figures 2d, 5a, 5b due to memory access constraint, with the first cycle for address generation and loading the pixel values from memory and the second cycle for the *sub* and *abs* operations, irrespective to the timing characteristics of the AFUs. Also, as in Section 4, arithmetic optimizations have been applied. The AFU in Fig. 5a implements the motion compensation task. The AFUs in Figures 2d, 5a, and 5b are the suggested architectural extensions for the examined video encoding kernels that could be added to typical embedded processors, media-processing ASIPs or as custom logic to configurable processors already in the market (Altera Nios-II, ARC, Xtensa).

6. Conclusions

In this paper, an application analysis and instruction generation flow is used for automatically identifying beneficial instruction-set extensions of embedded processors for the case study of motion estimation algorithms. For this reason, a prototype instruction generation engine allowing multi-dimensional design space explorations of MaxMISO-generated custom instructions has been implemented. We have performed automated explorations regarding the maximum number of input operands, instruction latency, as well as node-inclusion and boundary-node constraints, constant multiplication and multi-operand addition in the search for optimal custom instructions and their corresponding application-specific functional units. It is concluded that only three AFUs are needed to accelerate a variety of motion-estimation related algorithms up to 3.5 times over a software-based implementation. In this context, manual optimizations, which are amenable to automation in future revisions of our tool, including user-defined state registers have been regarded with positive results, while exploiting narrow bitwidth operands in combination with SIMD parallelism, would have an even more profound impact on application acceleration.

References

[1] M. Gschwind, “Instruction Set Selection for ASIP Design”, Proc. of the 7th Symposium on Hardware/Software Codesign, Rome, Italy, May 1999, pp. 7-11.

[2] ARC homepage: http://www.arccores.com

[3] R. Gonzalez, “Xtensa: A configurable and extensible processor”, IEEE Micro, Vol. 20, No. 2, pp. 60-70, Mar.-Apr. 2000.

[4] LEON processor homepage: http://www.gaisler.com

[5] K. Keutzer, S. Malik, and A.R. Newton, “From ASIC to ASIP: The Next Design Discontinuity”, Proc. of the 20th Int. Conf. on Computer Design, Freiburg, Germany, Sep. 2002, pp. 84-90.

[6] J. Fritts, “MediaBench II”, Presented at the 2004 Workshop on Media and Signal Processors for Embedded Systems and SoCs, Washington, D.C., USA, Sept. 2004.

[7] C. Lee, M. Potkonjak, and W.H. Mangione-Smith, “MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communications Systems”, Proc. of the 30th IEEE/ACM Symposium on Microarchitecture, Research Triangle Park, North Carolina, USA, Dec. 1997, pp. 330-335.

[8] MachSUIF homepage: http://www.eecs.harvard.edu/hube/research/machsuif.html

[9] L. Pozzi, N. Vuletic, and P. Ienne, “Automatic Topology-based Identification of Instruction-Set Extensions for Embedded Processors”, Technical report CS01/377, Processor Architecture Laboratory, Swiss Federal Institute of Technology Lausanne, Dec. 2001.

[10] P. Castro, E. Borin, R. Azevedo, and G. Araujo, “Looking for Instruction Patterns in the Design of Extensible Processors”, Proc. of the 3rd Workshop on Application Specific Processors, Stockholm, Sweden, Sep. 2004.

[11] P. Yu, and T. Mitra, “Characterizing Embedded Applications for Instruction-Set Extensible Processors”, Proc. of the 41th Design Automation Conference, San Diega, CA, USA, June 2004, pp. 723-728.

[12] N. Clark, H. Zhong, W. Tang, and S. Mahlke, “Automatic Design of Application Specific Instruction Set Extensions through Dataflow Graph Exploration”, International Journal of Parallel Programming, Vol. 31, No. 6, pp. 429-449, Dec. 2003.

[13] F. Sun, S. Ravi, A. Ragnunathan, and N.K. Jha, “Custom-Instruction Synthesis for Extensible-Processor Platforms”, IEEE Trans. on Computer-Aided Design, Vol. 23, No. 2, Feb. 2004, pp. 216-228.

[14] D. Goodwin, and D. Petkov, “Automatic generation of application specific processors,” Proc. of the Int. Conf. on Compilers, Architectures and Synthesis of Embedded Systems, San Jose, California, USA, Oct.-Nov. 2003, pp. 137-147.

[15] E. Borin, F. Klein, and N. Moreano, R. Araujo, and G. Araujo, “Fast Instruction Set Customization”, Proc. of the 2nd Workshop on Embedded Systems for Real-Time Multimedia, Stockholm, Sweden, Sep. 2004.

[16] Pattlib homepage: http://www.lsc.ic.unicamp.br/pattlib/

[17] R. Schreiber, S. Aditya, S. Mahlke, V. Kathail, B.R. Rau, D. Cronquist, and M. Sivaraman, “PICO-NPA: High-Level Synthesis of Nonprogrammable Hardware Accelerators”, Journal of VLSI Signal Processing, Vol. 31, 2002, pp. 127-142.

[18] ARM Ltd., MOVE Coprocessor Technical Reference Manual, Apr. 2002.

[19] P. Kuhn, Algorithms, Complexity Analysis and VLSI Architectures for MPEG-4 Motion Estimation, Kluwer Academic Publishers, Boston, 1999.

[20] N. Kavvadias and S. Nikolaidis, “Application Analysis with Integrated Identification of Complex Instructions for Configurable Processors”, Proc. of the 14th Int. Workshop on Power and Timing Modeling, Optimization and Simulation, Santorini, Greece, Sep. 2004, pp. 633-642.

[21] SUIF homepage: http://suif.stanford.edu/suif/suif2/

[22] Processor Architecture Laboratory at EPFL homepage: http://lapwww.epfl.ch/dev/

[23] P. Biswas, V. Choudhary, K. Atasu, L. Pozzi, P. Ienne, and N. Dutt, “Introduction to Local Memory Elements”, Proc. of the 41st Design Automation Conference, San Diego, California, USA, June 2004, pp. 729-734.

[24] R. Bernstein, “Multiplication by integer constants”, Software – Practice and Experience, Vol. 16, No. 7, July 1986, pp. 641-652.

[25] P. Briggs, and T. Harvey, “Multiplication by integer constants”, Technical report, Rice University, July 1994.

[26] S. Vassiliadis, J. Philips, and B. Blaner, “Interlock Collapsing ALUs”, IEEE Trans. on Computers, Vol. 42, No. 7, July 1993, pp. 825-839.

[27] J.-L. Beuchat, “A VHDL Library for Integer and Modular Arithmetic”, User Manual for the intmodlib VHDL source code, Version 0.1, Sep. 2004. Homepage: http://perso.ens-lyon.fr/jean-luc.beuchat/ArithLib

[28] International Organization of Standardization, Working Group on Coding of Moving Pictures and Audio, MPEG-4 Video Verification Model Version 18.0, Pisa, January 2001.

[29] MPEG-4 shape encoder project homepage: http://www.ece.cmu.edu/~ece796/project99/12/final/

[30] D.C. Burger and T.M. Austin, “The SimpleScalar Tool Set, Version 2.0”, University of Winsconsin-Madison, Computer Sciences Department, Technical report #1342, June 1997.

[31] L.M. Po, C.K. Cheung, and C.H. Cheung, Various Advanced Motion Estimation Research Development Package, Aug. 2000. Homepage: http://www.ee.cityu.edu.hk/~lmpo/publications/

[32] T. Koga, K. Iinuma, A. Hirano, Y. Iijima, and T. Ishiguro, “Motion compensated interframe coding for video conferencing”, Proc. of the National Telecommunications Conference, New Orleans, LA, Nov. 1981, pp.G5.3.1-G5.3.5.

[33] L.M. Po, and W.C. Ma, “A novel four-step search algorithm for fast block motion estimation”, IEEE Trans. on Circuits and Systems for Video Technology, Vol. 6, No. 3, pp. 313-317, June 1996.

[34] R. Li, B. Zeng, and M. L. Liou, “A new three-step search algorithm for block motion estimation”, IEEE Trans. on Circuits and Systems for Video Technology, Vol. 4, No. 4, pp. 438-442, Aug. 1994.

[35] L.-K. Liu, and E. Feig, “A Block-based Gradient Descent Search Algorithm for block-based motion estimation in video coding”, IEEE Trans. on Circuits and Systems for Video Technology, Vol. 6, No. 4, Aug. 1996, pp. 419-422.

[36] B. Liu, and A. Zaccarin, “New fast algorithms for the estimation of block motion vectors”, IEEE Trans. on Circuits and Systems for Video Technology, Vol. 3, No. 2, pp. 148-157, Apr. 1993.

[37] C.D. Bei, and R.M. Gray, “An improvement of the minimum distortion encoding algorithm for vector quantization”, IEEE Trans. on Communications, Vol. 33, pp. 1132-1133, Oct. 1985.

[38] J.Y. Tham, S. Ranganath, M. Ranganath, and A.A. Kassim, “A novel unrestricted center-biased diamond search algorithm for block motion estimation,” IEEE Trans. on Circuits and Systems for Video Technology, Vol. 8, pp. 369-377, Aug. 1998.

[39] M. Budiu, N. Sakr, K. Walker, and S.C. Goldstein, “BitValue Inference: Detecting and Exploiting Narrow Bitwidth Computations”, Proc. of the European Conference on Parallel Processing, Munich, Germany, 2000, pp. 969-979.

[40] S. Mahlke, R. Ravindran, M. Schlansker, R. Schreiber, and T. Sherwood, “Bitwidth Cognizant Architecture Synthesis of Custom Hardware Accelerators,” IEEE Trans. on Computer-Aided Design, Vol. 20, No. 11, pp. 1355-1371, Nov. 2001.