A Flexible Instruction Generation Framework for Extending Embedded Processors

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

***Abstract*— Modern platform-based design involves the domain-specific extension of embedded processors to fit customer requirements. To accomplish this task, the possibilities offered by recent extensible processors for their instruction set and microarchitectural customization have to be exploited. In this paper, a design approach that encapsulates automated workload characterization and highly-controllable instruction and application-specific functional unit (AFU) generation is utilized for fast extension space exploration of embedded processors. It is proved that a relatively small number of unique AFUs is needed in order to support embedded applications from the MiBench and Powerstone suites. It is possible to achieve 1.8× to 6.8× performance improvements although further possibilities such as subword parallelization are not currently regarded.**

I. Introduction and related work

Modern design flows for embedded processors (e.g. for consumer applications) are subjected to several constraints stemming out of diverse and often conflicting requirements for programmable systems-on-chip (SoCs): low power consumption, performance in a given application domain, code size and acceptable overall system cost.

The challenge of delivering the optimum balance between efficiency and flexibility can be met with the utilization of customizable processors [1], [2] adhering to the configurable/extensible processor paradigm. Configurability lies in tuning architectural parameters (e.g. cache sizes) while extensibility of a processor comes in modifying the instruction set architecture by adding custom instructions. There exist two basic themes for architecture extension: tight integration of custom functional units and storage [3] or loose coupling of hardware accelerators to the processor through a bus interface [4]. Focusing on the former case, this may require the introduction of custom units to the execution stage(s) of the processor pipeline and this should be accounted in the architecture template of the processor.

Last years, a number of research efforts have regarded the automated application-specific extension of embedded processors [5]–[9]. A few open instruction generation frameworks exist [10]; an advantage of their work being delivering a format for storing, manipulating and exchanging instruction patterns. In order to use their pattern library (Pattlib), the potential user should adapt his compiler for generating and manipulating patterns in the cumbersome GCC RTL (Register Transfer Language) intermediate representation.

Application-specific instructions have been generated for the Xtensa configurable processor [7] that may comprise of VLIW (Very Long Instruction Word), SIMD (Single-Instruction Multiple-Data) or fused (chained) RTL operations. However, as induced by the architecture template of Xtensa, control-transfer instructions (*cti*) are not considered to be included in the resulting complex instructions. A sophisticated framework for the design of tightly-coupled custom coprocessing datapaths and their integration to existing processors has been presented in [6]. While providing a complete solution to programmable acceleration, their work still has some drawbacks: the possibility of direct communication to fast local data memory is excluded and for this reason, beneficial addressing modes cannot be identified. In [8] a multiple-output instruction generation algorithm is presented which selects maximal-speedup convex subgraphs for each basic block data-dependence graph (DDG), with worst case exponential complexity, while [5] added path profiling to extend beyond basic block scope. An important conclusion was that useful instruction identification scope does not extend further than 2 or 3 consecutive basic blocks. Still, memory operations are not regarded in the formation of complex instructions, while pattern identification can only take place post register allocation.

In this paper, a custom instruction/AFU generation and selection prototype framework is presented. The method is based on an extension of the MaxMISO algorithm [9], which identifies the maximal non-overlapping connected subgraphs of the DDGs of basic blocks in application programs that produce a single computation result. In this context, two different types of node constraints named *node-inclusion* and *boundary-node* constraints are introduced while additional parameterization options of the instruction generation process have been integrated in the algorithm (selectable number of nodes, inputs, arithmetic and other optimizations).

II. Instruction/AFU generation framework

Our instruction generation approach has been implemented in context of a new framework named IAG (*Instruction/AFU Generation* framework) illustrated in Fig. 1. A CDFG IR is used for representing control and data-dependence, which is based on the SUIF/MachSUIF SUIFvm IR [11], [12]. The resulting IR can use *SUIFvm*, *SUIFrm* (introducing allocable

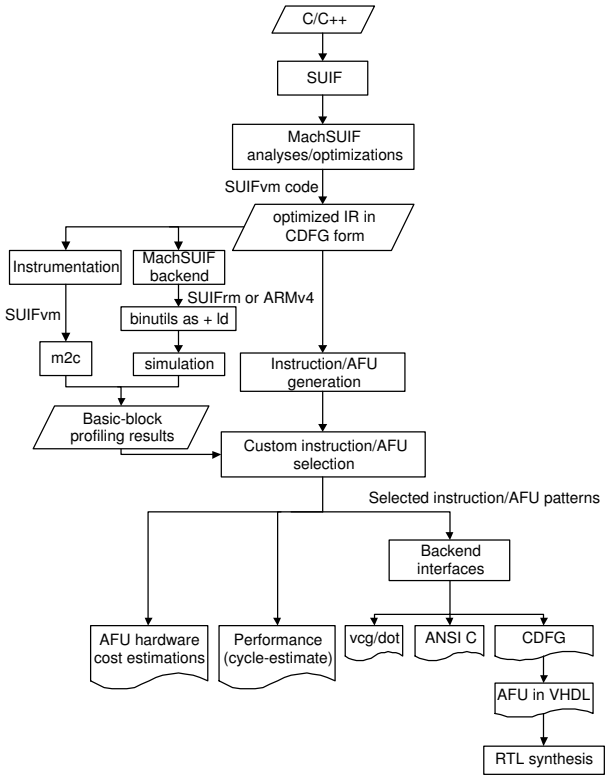


Fig. 1: The IAG framework.

resources to *SUIFvm*) or *ARMv4* [13] instruction nodes. For *SUIFvm* and *SUIFrm*, complete procedure entry and exit sequences have not been inserted at this stage, since stack frame layout is highly processor dependent. However, an instruction-accurate ArchC [14] model for the *SUIFrm* architecture has been implemented that supports input/output argument register banks to which the corresponding operands are mapped by appropriate pseudo-instructions. *ARMv4* CDFGs carry machine dependencies that could be exploited for porting application binaries to revised ARM ISAs or synthesizing a hardware-assisted RTOS for a given embedded processor. The MachSUIF passes perform either IR analyses or optimizations to optimize the *SUIFvm*/ARM assembly.

The pattern generation process takes place on the resulting IR, followed by custom instruction selection which is also implemented within IAG. The most important features of IAG's pattern identification and generation engine involve the enforcement of controlling parameters/constraints that can be divided into three categories: a) maximum value parameters, b) architecture template parameters and c) configuration and interoperability settings. Table I shows the notations and the descriptions for the configuration parameters that are available to the IAG framework user.

The two types of node constraints are: Type-A or *boundary-node* and Type-B or *node-inclusion*. A Type-A constraint prohibits growing an instruction cluster beyond the specified instruction, typically applied to force the generation of complex addressing modes. A Type-B constraint will not permit

TABLE I: Most important configuration parameters in IGF.

Configuration parameter	Description
Maximum value parameters	
ni	Max. input operands per MISO
nn	Max. num. instructions comprising a MISO
limit-[*a*|*i*|*c*]	Limit on total area (a), number of custom instructions (i) or max. cycles (c) per MISO
Architectural parameters	
constr-[*a*|*b*] {*list*}	Boundary node or node inclusion constraints on the given opcode list
isa-context	ISA context (SUIFvm, SUIFrm or ARMv4)
const-mul	Single constant multiplication optimization
Various configuration and interoperability settings	
isom-{*c*}-{*sc*}-{*ssc*}	Graph or graph-subgraph isomorphism ({c}) employing selected method ({sc}) on the generated patterns. {ssc} defines matching attributes
isel[-{*pf*}]	Custom instruction selection (for priority function {pf} in case of greedy selection)
gen-{*bend*}-{*clst*}	Export CDFGs of a cluster type {clst} (basic blocks or MISOs) to a backend format (vcg [15], dot [16], tac, and cdfg [17]).

the inclusion of the specified instruction in the MaxMISO under build, usually applied to *cti* instructions.

Regarding custom instruction selection, an optimal (formulated as 0-1 knapsack problem and solved via dynamic programming) and a greedy method based on predefined priority metrics have been implemented. The supported graph isomorphism and graph-subgraph isomorphism algorithms are part of the VFLib2 graph matching library [18]. Applying graph isomorphism identifies the unique custom instruction patterns, while applying graph-subgraph isomorphism is used for identifying the patterns corresponding to unique AFUs, servicing a subset of generated instructions. The basic two ISA configurations are: single opcodes and resource classes. Different instructions with opcodes of the same class can be matched and considered to be implemented on the same basic resource.

The IAG framework also provides interfaces to external visualization (*vcg, dot*) and backend (three-address C, CDFG [17]) engines for translating the generated custom instructions/AFUs for visualization, co-simulation, hardware estimation, or scheduling purposes.

III. Automated design space exploration scenarios for embedded processor extension

At this point, we will evaluate the proposed approach for instruction-set extension of embedded processors. For the experiments we have evaluated a set of embedded applications consisting of 3 cryptographic applications (*gost, rc5, sha*), 6 media-oriented applications (*3ss, adpcm_dec, adpcm_enc, jpeg_decode, mpeg4_senc, susan*), and 3 control-dominated applications (*dijkstra, patricia, stringsearch*). Most of these applications (7 out of 12) have been collected from the MiBench embedded benchmarks [19], while *jpeg_decode* is included in Powerstone [20]. All applications were compiled to SUIFvm code.

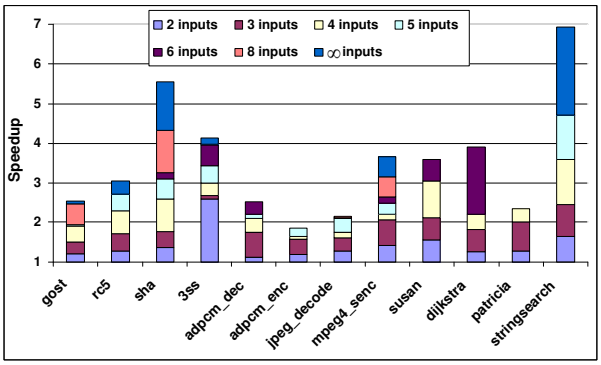


Fig. 2: Application speedup for different number of inputs.

The following figures (Fig. 2- 4) present the experimental results (execution cycles and area) on generation of custom instructions/AFUs under 3 exploration scenarios: a) instruction generation for different number of inputs, b) different node-related constraints, and c) greedy instruction selection under different priority functions.

A. Instruction generation for different number of inputs

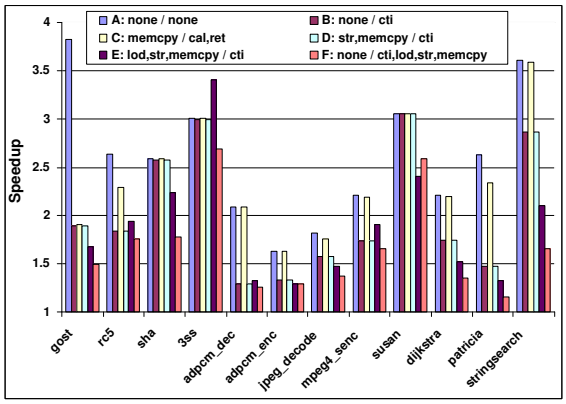
Fig. 2 shows the estimated speedups for the examined applications over the set of different number of inputs: $N_i = \{2, 3, 4, 5, 6, 8, \infty\}$ under the node constraints of case C, defined in Fig. 3. Area and delay metrics are normalized to the values for a 32×32-bit single-cycle multiplier returning a 64-bit result (not truncated).

By observation of Fig. 2 it can be deduced that the relationship among the achieved speedups and increasing the input size limit is monotonous but the amount of incremental speedup is completely application-dependent. This conclusion supports the necessity of detailed exploration in order to extract the speedup potential for each application.

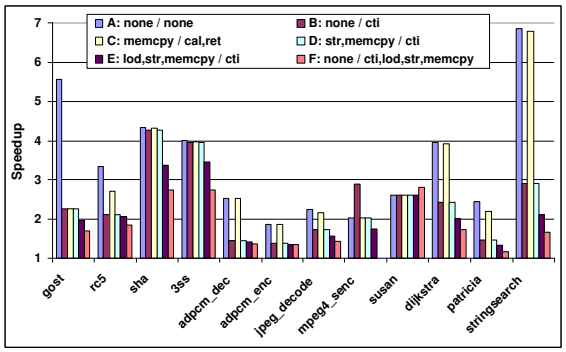
B. Instruction generation under different node constraints

As mentioned in Section II, our method is able to adapt to different architecture templates by imposing required limitations on selectable operation types during MISO formation. For example, a node-inclusion constraint should be forced on *cti* instructions for processor templates that do not permit altering the control transfer mechanisms. If allowed, the instruction fetch path is modified and the effect on 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. Fig. 3 presents results on studying the effect of node constraints on instruction generation for the examined benchmarks. Each node constraint is given in the form: {Type-A list}/{Type-B list} where 'Type-L list' denotes the opcode subset subjected to the specified constraint.

In Fig. 3 it is observed that the completely unconstrained case performs better in average to the other alternatives with an estimated average of 28%, with the exceptions of *3ss* (Fig. 3(a)), and *mpeg4_senc, susan* (Fig. 3(b)) for which the optimal performance metrics are obtained for cases E, B and F, respectively. This outcome can be interpreted by the fact



(a) N_i = 4



(b) N_i = 8

Fig. 3: Application speedup for alternative node constraints.

that *ctis* have the lowest priority in a topologically-sorted instruction ordering in a basic block. Thus, it is often for custom MISO instructions to be grown with a *cti* instruction as their sink node, with more data processing instructions being excluded from the MISO due to the input size limit being reached. If the *cti* was omitted, it is possible, that the algorithm would select larger computation structures for the same input size limit. Also, the introduction of complex *ctis* (case A) provides a speedup improvement of 25.9% over case B. However, it should be noted that the inclusion of call/return instructions may incur significant performance overhead in terms of storage resources (e.g. PC return stack, register windows) that cannot be directly accounted in the current version of the instruction/AFU generation tool.

C. Custom instruction selection

For implementing a greedy solution to the custom instruction selection problem, the key idea is to assign priorities to the custom instruction patterns and the more proficient instances are chosen by starting with the highest prioritized one. We have used the following two priority functions: (1) Cycle gain : $Priority(\sum_j C_{i,j}) = \sum_j \{P_{i,j} \times f_{i,j}\}$, that forces for best performance regardless AFU area requirements and (2) Cycle gain/Area : $Priority(\sum_j C_{i,j}) = \sum_j \{(P_{i,j} \times f_{i,j})\}/A_i$, where $C_{i,j}$ denotes the i-th candidate instruction with j different instances in the entire program, $f_{i,j}$ the basic block execution frequency metric associated with the specific

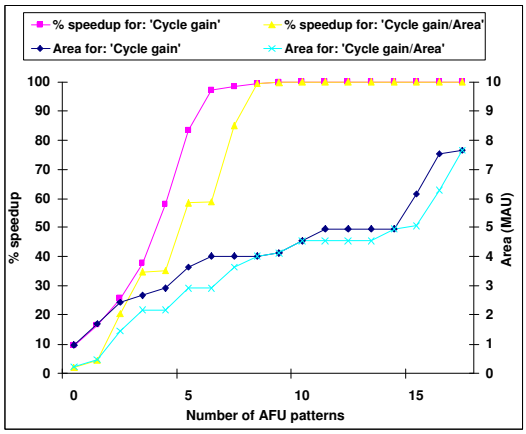


Fig. 4: Custom instruction selection under priority metrics for *sha* ($N_i = \infty$).

instance, and A_i the area cost for the candidate. These priority functions force different objectives: function (1) maximizes performance gain for each isomorphic candidate over the entire program when area is not an issue while (2) quantifies the available area budget as well.

A summary of the measurements for the application set is given in Table II. Taking *sha* for example, although tens of candidate instructions are identified, only a few (9 and 22 for achieving 95% and total maximum speedup, respectively) contribute significantly to their execution time for either priority function. The number of required extension instructions for reaching the 95% speedup levels ranges from 3 (*susan*) to 16 (*jpeg_decode*), while the area requirement is less than 5.5 multiples of a 32-bit array multiplier for all applications with the exception of *mpeg4_senc* which demands 8.4 multiplier area units (*MAU*).

Finally Fig. 4 directly compares the pros and cons for the priority functions used in the custom instruction selection process. For the *sha* application, there exists a profound speedup benefit for priority function 'Cycle gain' against 'Cycle gain/Area', since any differences in area occupation are relatively small while 2 less instructions need to be selected in order to obtain 95% of overall speedup for the former objective.

IV. Conclusions

In this paper, an instruction and application-specific functional unit (AFU) generation flow is presented for automatically identifying beneficial architectural extensions of embedded processors. For this reason, a prototype instruction generation engine allowing multi-dimensional design space explorations of custom instructions has been implemented. We have performed automated explorations regarding the maximum number of input operands, different node-related constraints, and instruction selection decisions in the search for optimal custom instructions and their corresponding AFUs for a diverse application set. It was found that speedups ranging from 1.8× to 6.8× can be achieved over implementing the same applications on an unaugmented base processor.

TABLE II: Speedup-AFU area for 'Cycle gain'/'Cycle gain/Area' (average case of $N_i = \{4, 8, \infty\}$).

Benchmark	0.95× max. speedup	Area (MAU)	At max. speedup	Area (MAU)
gost	8/10	1.827/1.471	16	2.532
rc5	8/9	3.087/3.087	23	7.771
sha	9/12	2.964/2.976	22	6.050
3ss	5/6	2.836/2.892	28	17.479
adpcm_dec	6/6	0.527/0.527	8	0.771
adpcm_enc	8/8	0.356/0.356	10	0.835
jpeg_decode	14/16	5.372/4.529	35	12.084
mpeg4_senc	9/12	8.420/5.893	50	30.221
susan	3/3	2.656/2.656	16	12.102
dijkstra	4/4	0.864/0.864	10	1.573
patricia	4/4	0.441/0.441	4	0.441
stringsearch	5/7	0.909/0.851	11	1.703

References

[1] ARC cores. [Online]. Available: http://www.arccores.com

[2] R. Gonzalez, "Xtensa: A configurable and extensible processor," *IEEE Micro*, vol. 20, no. 2, pp. 60–70, March-April 2000.

[3] Altera Nios-II home page. [Online]. Available: http://www.altera.com/products/ip/processors/nios2/

[4] Xilinx home page. [Online]. Available: http://www.xilinx.com

[5] P. Yu and T. Mitra, "Scalable custom instructions identification for instruction-set extensible processors," in *Proc. Int. Conf. on Compilers, Architectures and Synthesis for Embedded Systems*, September 2004.

[6] N. Clark, J. A. Blome, M. L. Chu, S. A. Mahlke, S. Biles, and K. Flautner, "An architecture framework for transparent instruction set customization in embedded processors," in *Proc. 32nd Int. Symp. on Computer Architecture*, Madison, Wisconsin, USA, June 2005, pp. 272–283.

[7] D. Goodwin and D. Petkov, "Automatic generation of application specific processors," in *Proc. Int. Conf. on Compilers, Architectures and Synthesis for Embedded Systems*, San Jose, California, USA, October 2003, pp. 137–147.

[8] K. Atasu, L. Pozzi, and P. Ienne, "Automatic application-specific instruction-set extensions under microarchitectural constraints," in *Proc. 40th ACM/IEEE Design Automation Conference*, Anaheim, California, June 2003, pp. 256–261.

[9] C. Alippi, W. Fornaciari, L. Pozzi, and M. Sami, "A DAG based design approach for reconfigurable VLIW processors," in *Proc. of the Design, Automation and Test in Europe Conf.*, Munich, Germany, March 1999, pp. 778–779.

[10] Pattlib. [Online]. Available: http://www.lsc.ic.unicamp.br/pattlib/

[11] SUIF. [Online]. Available: http://suif.stanford.edu/suif/suif2/

[12] Machine-SUIF research compiler. [Online]. Available: http://www.eecs.harvard.edu/hube/research/machsuif.html

[13] G. Théoduloz and D. S. Gracia, ARM backend for Machine SUIF. [Online]. Available: http://lapwww.epfl.ch/dev/

[14] The ArchC resource center. [Online]. Available: http://www.archc.org

[15] G. Sander, "Graph layout through the VCG tool," in *Proc. DIMACS Int. Workshop on Graph Drawing*, Berlin, Germany, 1994, pp. 194–205.

[16] E. R. Gansner, E. Koutsofios, S. C. North, and K.-P. Vo, "A technique for drawing directed graphs," *IEEE Trans. on Software Engineering*, vol. 19, no. 3, pp. 214–230, May 1993.

[17] CDFG toolset. [Online]. Available: http://poppy.snu.ac.kr/CDFG/cdfg.html

[18] P. Foggia, *The VFLib Graph Matching Library*, 2nd ed., March 2001. [Online]. Available: http://amalfi.dis.unina.it/graph/db/vflib-2.0

[19] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, "MiBench: A free, commercially representative embedded benchmark suite," in *Proc. of the 4th annual IEEE Int. Wshp. on Workload Characterization*, December 2001.

[20] L. H. Lee, W. Moyer, and J. Arends, "Instruction fetch energy reduction using loop caches for embedded applications with small tight loops," in *Proc. Int. Symp. on Low Power Electronics and Design*, San Diego, CA, August 1999.