Macro Instruction Synthesis for Simple Embedded RISC Processor

CS252 class project report

Yunjian (William) Jiang and Pinhong Chen
EECS, UC Berkeley

Abstract

Synthesis of application specific macro instructions for an embedded processor is an efficient realization of computation on a limited silicon area. Commercial solutions of configurable microprocessors lack the automation of generating specific instructions for a target application domain. In this project, we explore algorithms and CAD solutions for automatically generating macro instructions from a set of benchmark programs and associated hardware/software supports. However, constrained by the limited time and effort, we base our ideas and experiments on a simple RISC processor core. The algorithm we are exploring is based on pattern enumeration for expression trees in the intermediate representation (IR) of an application program. During the enumeration we use annotated profiling information from our simulation results. Our target is to automate the process of generating the code generator, assembler and simulator, since it is a very simple process core(3000 gates in 0.25um with 200MHz).

1 Introduction

As the explosion of mobile and embedded computing in the consumer market, e.g. cell phones, palm pilots, palm-top computers and multimedia devices, application specific instruction processors (ASIP) are becoming widely used and designed just for a limited set of applications. The research topics associated with the synthesis of ASIPs are basically application modeling, architecture configuration and architecture mapping. There are different levels of architecture configuration, including (1) architecture organization level, e.g. VLIW or superscalar, the number and types of execution units; (2) instruction level, e.g. the instruction set architecture and the control/data path; (3) parameterizable decisions, e.g. register files, cache size and associativity, etc. In this project, we are particularly interested in exploring the space of instruction set architecture.

In this methodology, we start from a simple RISC processor core. Given the set of applications being targeted, we then generate the set of macro instruction which would speed up the performance and reduce code size significantly. Also, a software tool-set, including code-generator, simulator and assembler, will be generated accordingly to utilize the generated instructions. The advantage is that no midification is required on the original algorithm or source code. The problems are now how to identify special instruction patterns, how to evaluate the profit of a particular macro instruction and how to implement the macro instruction with hardware and software support. These are the questions we are trying to solve.

A similar scenario is been commercialized by Tensilica [1], which would allow users to configure their own processor, including special instructions, and build the associated software tool-set. However the extended instructions have to be generated manually by the designer and the source code has to be modified manually in order to use any newly designed instruction. A related research in [2] addresses the instruction set synthesis problem for DSP processors. They target a special DSP micro architecture and address a problem of building the entire instruction set from application. They formulate the problem as a subset-sum problem and make use of a combinatorial solver. It is more of a low level hardware synthesis approach.

As will be elaborated in the remaining sections, the contribution of this project is twofold: first the register transfer level (RTL) tree pattern enumeration for instruction identification and second the table-driven assembly tool-set automation.

2 RTL pattern enumeration

Based on the expression tree-based intermediate representation (IR) of a software application, we construct a library of profitable macro instructions subject to re-

source constraints. The common flow of code generation in an embedded compiler is the following:

1. Code generation by pattern matching;
2. Register allocation;
3. Macro instrction expansion;
4. Assemblying;

In code generation phase, a high level machine independent IR from a compiler front-end, like SUIF-IR, is mapped into intruction patterns available in the ISA' pattern library. This is called a register transfer level (RTL) expression tree, which utilized symbolic registers. After register allocation, macro instructions are expanded into real assembly codes provided by the architecture. Finaly, the assemblying phase outputs binary machine code for simulation.

The expression tree is enumerated for identifying complex instruction patterns as candidates for new macro instructions. We have discovered that in order to generate meaningful instruction patterns and automate the process of code generation, assembling and simulation, we have to conduct the pattern enumeration at the register transfer level after the code generation phase.

There has been research on instruction enumeration at machine independent IR level. The disadvantages are: the estimation of performance and code size saving is crude; it is hard to integrate profiling information to identify hot-spot. After register allocation, there is a one-to-one mapping between the RTL expression tree and macro assembly code. Thus performance and code size can be precisely predicted. Furthermore, profiling information can be back-annotated from our simulation tool.

For instruction enumeration, we apply the following algorithm for each expression tree in each basic block.

```
Traverse each node i in post-order
  If i is leaf node
    Store(Instr,OP(i,NULL),cost(OP(i));
    Continue;
  Normalize sub-tree order;
  Foreach u in Instr(left-node)
    Foreach v in Instr(right-node)
      cost(i) = OP(i)+cost(u)+cost(v);
      save(i) = save(i)+save(u)+save(v);
      Store(Instr,Op(i,u,v),cost,save);

Select the best instruction from Instr();
End
```

At each node of the expression tree: the sub-trees are normalized; the performance gains and hardware costs are estimated based on the information collected from sub-tree nodes; then the new instruction pattern and this information are hashed. Finally, the best instructions are selected from all patterns constructed throughout the program. There is no hard limit on the types of instructions to be identified. Through the experiment, we actually found very complex instruction patterns with a number of arithmetic operations and memory accesses. These will be selected and implemented if the performance saving outweigh the hardware costs.

After code generation, the expression tree is built based on *non-terminals* in the pattern matching grammar. In particular, we retargeted the code-generator generator *olive* from [3] to our RISC8 instruction set. The *non-terminals* includes 8-bit registers `reg`, 8-bit accumulator `acc`, 16-bit address registers `areg` and different constants appearing as immediate values `const08`, *el at.*. The identified instruction patterns directly correlate to the instruction patterns to be specified in the *olive* grammar. This greatly alleviates the effort of reconfiguring the code-generator to take into account the new instruction. We have implemented a PERL tool which automatically builds the *olive* pattern from the library of new identified instructions. The following is an example of the *olive* patterns generated from a new instruction `ADDCON`.

```
acc : nAND (acc , uconst08)
{
$cost[0].cost=1+$cost[2].cost+$cost[3].cost;
$cost[0] = COST_INFINITY;
} = {
AsmRegister* r0 =new AsmRegister($action[2]());
AsmOperation* ao=new AsmOperation( aANDCON,
                 new AsmImmediate($3->value()),
                 NULL);
cil->Emit( new CompactedInstruction(ao,NULL) );
return r0;
};
```

3 Assembly Level Instruction Pattern Enumeration

It is also possible to recover data dependency at assembly code level and identify new instructions. The following is an example of a fragment of assembly code and the corresponding RTL expression tree shown in Figure 1(a).

```
        ldi  r1,0x89
        lda  r0,variable_x
```

```
and  r1
lda  r2,variable_y
add  r2
```

r0 is the accumulator denoted by "acc", r1 and r2 are general registers; ldi and lda are instructions for loading a constant and a variable from memory, respectively. An instruction pattern usually does not rely on any specific data storage, thus we eliminate intermediate registers and obtain a more concise expression tree as shown in Figure 1(b). The reduced expression tree is ready for a new instruction candidate. It will be back-annotated with the execution profiling information in the instruction selection phase.

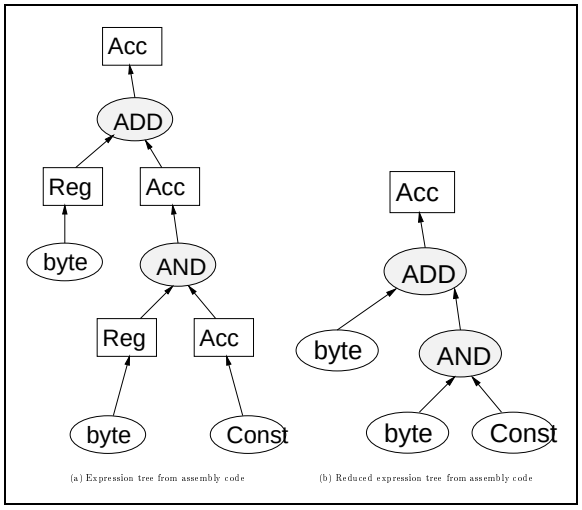

Figure 1: Assembly pattern recovery

4 Table-Driven Assembly tools

After the set of special instructions are identified, the software support for implementing the instructions involves

1. Configuring the code-generator with the new pattern;
2. Assigning an opcode for the new instruction;
3. Annotating the assembler with such opcode;
4. Augmenting the simulator with the functionality of the new instruction.

We are able to achieve automating all these processes with the concept of table driven assembly tool-set configuration. Figure 2shows how these programs interact with each other.

The key in the automation of these tools is what we call a table driven technique. We implement a complex PERL data structure to act as the driving mechanism to interpret a new instruction in various assembly tools.

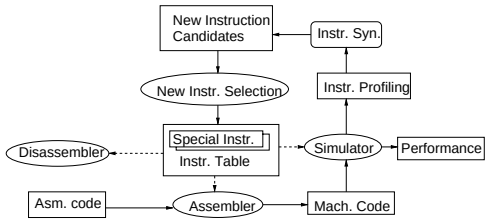

Figure 2: Automation of New Instruction Generation

For example, the table entry for a new instruction is shown below:

```
@new_ins=(
    'andcon'=>{
          pattern=>'imm8',
          code=>['01111100','$imm8'],
          decode=>'$imm8=$memory[$pc++]',
          sim=>'$R[0]=$R[0] & $imm8;',
          cycles=>1
    },
);
```

Here, an instruction called andcon is defined with the following features:

1. *Operand*: a single 8-bit immediate value, "imm8";
2. *Op-code sequence*: "01111100"(binary value) followed by the immediate operand;
3. *Decoding method*: interpreting the next memory data byte as the immediate operand value,
4. *Simulation method*: "reg&imm8"
5. *Number of cycles*: one.

PERL codes are used extensively to achieve efficient reconfigurability. Actually, the assembler, simulator, disassembler, instruction selector, and new instruction table generator are all coded in PERL for seamless integration.

5 Profiling Information Back Annotation

Back-annotation of instruction profiling information is an important step for instruction selection. The machine code generation process is divided into four phased, as show in Figure 3. Code generation converters the expression tree into RTL trees using symbolic

register. The instruction patterns used at this step include macro instructions accounting for complex operations like 16-bit arithmetics, which are common in normal architectures but not implemented here in RISC8. These are expanded into real RISC8 assembly codes in the macro expansion phase. The profiling results from the simulation of binary machine codes need to traverse back through all the four phases with different representations. Therefore, in each of these steps we generate a mapping file to describe the relationship between upper-level and lower-level codes. The instruction selector maps the profiling information all the way back to the code generation phase to annotate each IR node at the RTL level.

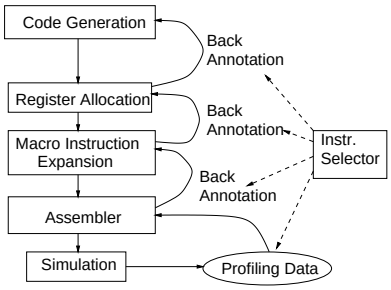

Figure 3: Back Annotation Flow of Profiling Information

6 Op-code Reuse

For a specific application, not each op-code may appear in the machine code. Namely, some of op-codes are never used throughout a particular application program. This appears in the cases where only a subset of the operations specified in the ISA are used; or the code generator does not generate the full spectrum of all instructions. We have studied some of the applications and find a very interesting result. The used op-codes are around one fifth of the total available opcodes for each application we experimented, as shown in Table 1. *This table implies that the op-codes can be reused to reduce decoding hardware cost.* The decoding logic can be smaller and faster. Furthermore, some functional units may be trimmed if it is not used in a specific application. However, the flexibility and functionality of the ISA is reduced. It may not be possible to adapt to other applications after op-code reuse.

Application	Opcodes
FIR	28
ADPCM	49
GSM	32
max7219	39
LCD4x20	40
PRN-IO	30

Table 1: Op-code Usage

7 Experimental results

We base our experiment on a publically available 8-bit RISC core [4]. In our implementation, we use SUIF [5] as our compiler front end; we retargeted SPAM [3] for the RISC8 processor, using the code-generator generator *olive* as our configurable code generator; we implemented the RTL pattern enumeration algorithm in C++, configurable RISC8 assembler and simulator in PERL.

We collected the following benchmark applications for our 8-bit configurable RISC processor as shown in Table 2. We are able to generate a set of interesting instructions for each applications. These are not shown due to space limit. We focus our attention on the GSM encoder and build some of the generated instruction. These are shown in Table 3. The performance gains and hardware cost estimations for each identified instruction are shown in Table 4. Comparing with the original RISC8 implementation, we are able to obtain about 40% reduction in the cycle count and about 30% reduction in code size with little hardware overhead.

8 Conclusion

In conclusion, our contribution of this project is that we experimented and implemented the flow of automatic macro-instruction identification and construction. This includes pattern enumeration at both RTL level and assembly level and the automatic configuration of code-generator, assembler and simulator. We have shown that the architecture space exploration at instruction level for a particular application can be achieved with a push-button solution by the concepts and techniques described in this report. This does not require any effort from the designers for making changes at the algorithm source level.

Apparently, performance gains come from configuration at all levels, including algorithms, hardware software partitioning, software implementations and processor architecture organizations. As mentioned in section 1, we only address the design space at instruction set architecture level and achieve a full automation flow for the exploration within basic blocks. The performance and code size gain through our implementation is 30%-40% without human effort. The comparison of

the potential performance gains from different implementation levels is out of the scope of this project.

Some of the future directions of this research are pointed out below:

1. The repetition of possible tree patterns is dependent on how the IR is generated, it is profitable to expand the solution space by applying tree rewriting techniques based on some transformation rules.

2. Type information can be incorporated in pattern construction process to pack multiple data operations into one instruction, or SIMD instructions.

3. Break the basic block limit and apply pattern construction across control structures. This is the most effective way for identifying instruction level and loop level parallelism.

References

[1] "Tensilica inc.." `http://www.tensilica.com`.

[2] H. Choi, J.-S. Kim, C.-W. Yoon, I.-C. Park, S.-H. Hwang, and C.-M. Kyung, "Synthesis of application specific instructions for embedded dsp software," *IEEE Trans on Computers*, 1999.

[3] "SPAM project ." `http://www.ee.princeton.edu/spam/`.

[4] "8-bit risc microprocessor core ." `http://www.geocities.com/microprocessors`.

[5] "SUIF project." `http://suif.stanford.edu`.

examples	*source*	*description*
ADPCM	MediaBench	16-bit voice compression and decompression
GSM	MediaBench	wireless communication voice encoder
PRN-IO	micro-controller	printer IO controller
LCD-4X20	micro-controller	LCD display controller
max7219	micro-controller	LEC display driver and controller

Table 2: Benchmarks used for macro instruction synthesis

Pattern Name	Instruction Pattern
Ins1	nASSIGN(addr16,nADD(nVAR(addr16),reg))
Ins2	reg=nADD(nVAR(addr16),reg)
Ins3	nASSIGN(addr16,nAND(nCON(imm8),nVAR(addr16)))
Ins4	reg=nAND(nCON(imm8),nVAR(addr16))
Ins5	reg=nADD(nVAR(addr16),nRORC(reg))
Ins6	reg=nADD(nVAR(addr16),nASR(reg))

Table 3: Instruction Pattern for Profiling Information

Total Saving	Each Saving	Instr. Count	Total ExeCyl	#Cycle	#Basic Instr.	#Reg	#AReg	#Mem Access	Pattern Name
547	3	16	1460	8	3	1	0	2	Ins1
338	1	30	1692	5	2	2	0	1	Ins2
336	4	6	840	10	4	0	0	2	Ins3
336	2	12	1176	7	3	1	0	1	Ins4
260	2	10	910	7	3	2	0	1	Ins5
260	2	10	910	7	3	2	0	1	Ins6

Table 4: Profiling Information