Synthesis of Instruction Sets for Pipelined Microprocessors[†]

Ing-Jer Huang and Alvin M. Despain
Advanced Computer Architecture Laboratory
Department of Electrical Engineering – Systems
University of Southern California
ijhuang@usc.edu, despain@usc.edu

Abstract

We present a systematic approach to synthesize an instruction set such that the given application software can be efficiently mapped to a parameterized, pipelined microarchitecture. In addition, the assembly code is generated to show how the application can be compiled with the synthesized instruction set. The design of instruction sets is formulated as a modified scheduling problem. A binary tuple is proposed to model the semantics of instructions and integrate the instruction formation process into the scheduling process. A simulated annealing scheme is used to solve for the schedules. Experiments have shown that the approach is capable of synthesizing powerful instructions for modern pipelined microprocessors. The synthesis algorithm ran with reasonable time and a modest amount of memory for large benchmarks.

1. Introduction

Microprocessors (reprogrammable processors) offer a flexible and low cost solution for embedded systems with complex algorithms or control intensive applications. The performance of a microprocessor-based system depends on how efficiently the application can be mapped to the hardware. One key issue determining the success of the mapping is the design of the instruction set, which serves as the interface between the hardware and application. How to design an instruction set that closely matches the characteristics of the hardware and of the application is an important design problem.

In this paper we present the problem formulation and the algorithm of a systematic approach [6] which synthesizes application-specific instruction sets for parameterized, pipelined microarchitectures, from given application benchmarks. The problem is formulated as a modified scheduling problem, with the micro-operations (MOPs) representing the application benchmark as the nodes to be scheduled, subject to several design constraints. Instructions are formed by an instruction formation process that is integrated into the scheduling process. The compiled code of the application is generated, using the synthesized instruction set. A simulated annealing scheme is used to solve for the schedule and the instruction set. The design issues addressed in this approach include: instruction utilization, instruction operand encoding, delay load/store and delay branches.

†. This work was supported by the ARPA under contract No. Rutgers 4-26385.

The rest of the paper is organized as follows. Section 2 reviews related work. Section 3 presents the models for the micro-architectures, instruction sets and application benchmarks. Section 4 and Section 5 describe the problem formulation and algorithm, respectively. Section 6 demonstrates our techniques with some experiments. Section 7 discusses the current status, limitations, and future directions.

2. Prior work

Most of the early work on automatic instruction set designs view the design problem as a design process independent to the hardware implementation. Instructions were not restricted to single-cycle instructions since multiple-cycle instructions can be supported through micro-programming (firmware). Without knowing the decode/control complexity, the focus was mainly in directly supporting high level languages or increasing the code density. The results were CISC-like instructions. These studies include Haney's [1], Bose's [2] and Bennett's [3] work. These techniques are not suitable for designing instruction sets for modern pipelined processors.

Sato et al. [5] propose an integrated design framework for application specific instruction set processors. This framework generates profiling information from a given set of application benchmarks and their expected data. Based on the profiles, the design system customizes an instruction set from a super set, decides the hardware architecture (derived from the GCC's abstract machine model), and the related software development tools. This framework is similar to our work in terms of the inputs and outputs of the design system; however, it is different from ours in terms of the machine model and the design method. They assume a sequential (non-pipelined) machine model, whereas we assume a pipelined machine with data-stationary control model. On the other hand, they generate instruction sets by selecting subsets from a super set, whereas we synthesize the instruction sets directly in order to find new and useful instructions for the given application domain.

Different from previous approaches, Holmer [4] focuses on generating instruction sets for pipelined micro-architectures with parameterized data paths. The parameters for a data path include the number of read/write register ports, memory ports, number of functional units and the cycle counts for memory operation. Our work builds on Holmer's results and improves the problem formulation and synthesis algorithms, in order to generate application-specific instruction sets and compiled codes for microprocessor-based embedded systems.

Another design problem that is close to the instruction set design problem is microcode compaction [10][11]. However, it does not has the concept of 'instruction set' and its goal is to reduce the number of cycles to execute a microprogram. Instruction set design has more complicated goal: optimize and trade off the instruction set size, the program size, and the number of cycles to execute a program.

3. Design models

3.1. Instruction sets

The instruction sets are assumed to be of fixed word length, which is specified by the designer. An instruction consists of fields. The fields are a combination of some field types. The field types and their bit widths are provided by the designer. Table 1 lists the specification of some instruction field types and their bit widths, taken from the BAM instruction set [13]. Each instruction has one opcode field, but the use of other fields is constrained only by the total number of bits needed by the operations in the instruction.

Instruction Field Type	Number of bits
instruction word	32
opcode	6
register (R)	5
tag (T)	5
displacement (D)	16
immediate (I)	16
relation operator (OP)	2

Table 1: Bit width specification for some instruction field types

The operands of instructions can be encoded to become part of the opcodes. There are two ways to encode operands. First, a specific value can be permanently assigned to an operand and becomes *implicit* to the opcode. Second, the register specifiers can be *unified*. For example, the instruction `inc(R)` 'R<-R+1' is obtained from the general instruction `add(R1,R2,Immed)` 'R_1<-R_2+Immed'. The facts of R_1=R_2 (unifying register specifiers) and Immed=1 (fixing an operand to a specific value which becomes implicit) are encoded into the opcode `inc`. Encoding operands saves instruction fields, and allows more MOPs to be packed into a single instruction, at the cost of possibly larger instruction set size, additional connections and hardwired constants in the data path.

3.2. Microarchitectures

The design style supported here is a parameterized, pipelined microarchitecture. The pipeline is controlled in a data stationary fashion [8]. It consists of stages for instruction fetch, instruction decode, register read, arithmetic/logic operation, memory access, and register write. The first two stages are identical to all instructions. The last four stages, the *instruction execution stages*, are dependent on the semantics of the instructions.

The target microarchitecture can be fully described by specifying the supported MOPs and a set of parameters. The supported MOPs describe the functionality supported in the microarchitecture, and the connectivity among modules in the data path. For example, the first two columns of Table 2 list some of the MOPs supported in the VLSI-BAM microprocessor [14] and their corresponding MOP type IDs.

The set of parameters describes resource allocation and timing of functional modules. The parameters include the number of register-file read/write ports, number of memory ports, number of functional units, the sizes of the register file and memory, and the delay cycles of memory access, functional units and control flow change.

Each MOP supported by the data path is assigned costs for the instruction format and hardware resources. The costs of the instruction format are the instruction fields required to operate the MOPs, including register index, function selectors, and immediate data. The hardware costs are the resources required to support the MOP. The hardware resources include read/write ports of the register file, memory ports, and functional units. The third and fourth columns in Table 2 lists the costs for the corresponding MOPs.

Type ID	MOP	Instruction Format Cost*	Hardware Cost†
rr	R_1 <- R_2	R_1, R_2	1 R, 1 W
rra	R_1 <- R_1 + R_2	R_1, R_2	2 R, 1 W, 1 F
rrai	R_1 <- Immed + R_2	R_1, R_2, I	1 R, 1 W, 1 F
ri	R_1 <- Immed	R_1, I	1 W
rm	R_1 <- mem(R_2)	R_1, R_2	1 R, 1 W, 1 M
rmd	R_1 <- mem(R_2 + Immed)	R_1, R_2, I	1 R, 1 W, 1 M, 1 F
mr	mem(R_1) <- R_2	R_1, R_2	2 R, 1 M
mi	mem(R_1) <- Immed	R_1, I	1 R, 1 M
mrd	mem(R_1 + Disp) <- R_2	R_1, R_2, D	2 R, 1 M, 1 F
mrad	mem(R_1 + Disp) <- R_2 + Immed	R_1, R_2, D, I	2 R, 1 M, 2 F
jd	pc <- pc + Disp	I	1 F

Table 2: MOP specification

*. Refer to the notation in Table 1.

†. Notation: 'R'=read port of register-file, 'W'=write port of register-file, 'M'=memory port, 'F'=functional unit. The number specifies how many of the resources are required.

3.3. Application benchmarks

Each application benchmark is represented as a group of weighted basic blocks. The weight indicates how many times the associated basic block is executed in the benchmark. Basic blocks are mapped to data/control flow graphs of MOPs that are supported by the target microarchitecture. Figure 1 shows an example of a basic block, which consists of six MOPs. The bold labels before the MOPs are their IDs. The solid arrows are data-related dependencies. The dashed arrows are control dependencies.

4. Instruction set design as a modified scheduling problem

The instruction set design problem can be formulated as a modified scheduling problem. The inputs of the problem are: application benchmarks represented in MOPs, constraints of the

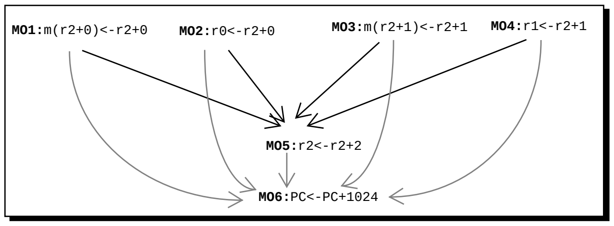

Figure 1. Data/control flow graph of MOPs of a basic block

instruction word width and instruction field widths, constraints of hardware resources, the number of delay cycles for branch/jump and memory MOPs, and the objective function. The MOPs of the benchmarks are scheduled into time steps, subject to various constraints to be discussed later. While scheduling MOPs into time steps, instructions are formed at the same time. Finally, the outputs of this problem formulation is a synthesized instruction set and compiled benchmarks.

Two schedules of the MOPs in Figure 1 are shown in Table 3 and Table 4, respectively. In the first column of the table are time steps, and in the second column are the IDs of the MOPs scheduled into the corresponding time step. In this example we assumed a one-cycle delay for the jump `MO6` MOP and zero-cycle delay for memory operations. The schedule in Table 3 is a serialized one, with seven cycles. There is one MOP in each time step. Note that there is a `nop` at the seventh cycle since `MO6` is scheduled as the last MOP. The schedule in Table 4 is a more compact one, with four cycles. Note that the delay slot of `MO6` is filled with `MO5` such that there is no need for a `nop`.

In the following subsections, we present several aspects of the scheduling problem.

4.1. Instruction formation: the binary tuple and its relation with scheduling process

The semantics of an instruction can be represented by a binary tuple <*MOPTypeIDs*, *IMPFields*>, where *MOPTypeIDs* is a list of type IDs (as shown in the first column of Table 2) of MOPs contained in the instruction, and *IMPFields* is a list of fields that are encoded into the opcode.

For example, the binary tuple for the instruction `add(R1,R2,Immed)` is `<[rrai],[]>`. The instruction contains one MOP 'R_1<-R_2+Immed' with the type ID `rrai`, which is represented by the list in the first argument of the tuple. Since no fields are encoded, the second argument of the tuple is an empty list. On the other hand, the binary tuple for the instruction `inc(R)`, an encoded version of the instruction `add(R1,R2,Immed)`as discussed in Section 3.1, is `<[rrai], [R1=R2,Immed=1]>`. The list in the second argument of the tuple specifies how the fields are encoded: The element `R1=R2` unifies the register specifiers R_1 and R_2 to the same register, and the element `Immed=1` fixes the immediate value permanently to the constant of one.

Instructions are generated from time steps in the schedule. Each time step corresponds to one instruction. The type IDs of the MOPs scheduled to the same time step are assigned to the first argument of the binary tuple for the instruction at the time step. The operand encoding specification, which is generated by an encoding process integrated into the scheduling process (described in Section 5), is assigned to the second argument of the binary tuple.

In Table 3 and Table 4, the columns under the header 'Instruction Semantics' and 'Instruction Fields' describe the semantics and field information of the instructions formed for the two schedules, respectively. The columns 'MOP type IDs' and 'Encoded fields' specify the binary tuples for the instructions. The RTLs for the corresponding MOP types are listed under the 'RTLs' column. The 'Inst Name' column assigns names to the generated instructions. The column 'Format' describes the

Schedule		Instruction Semantics				Instruction Fields		Costs	
Time step	MOP IDs	RTLs	MOP type IDs	Encoded fields	Inst. name	Format	Field values	Hardware cost*	Inst. word width
1	mo1	m(R_1+D_1)<-R_2+D_2	mrad		inst1	R_1, R_2, D_1, D_2	r2, r2, 0, 0	2R, 1M, 2F	48
2	mo2	R_1<-R_2+I	rrai	I=0	inst2	R_1, R_2	r0, r2	1R, 1W, 1F	16
3	mo3	m(R_1+D_1)<-R_2+D_2	mrad	D_1=D_2, R_1=R_2	inst3	R_1, D_1	r2, 1	1R, 1M, 1F	27
4	mo4	R_1<-R_2+I	rrai		inst4	R_1, R_2, I	r1, r2, 1	1R, 1W, 1F	32
5	mo5	R_1<-R_2+I	rrai		inst4	R_1, R_2, I	r2, r2, 2	1R, 1W, 1F	32
6	mo6	pc<-pc+D	jd		inst5	D	1024	1F	22
7	nop		nop		inst6				6

Table 3: Schedule I for the MOPs in Figure 1 and the resulted instructions

*. Refer to the foot note of Table 2 for the meaning of the notations.

Schedule		Instruction Semantics				Instruction Fields		Costs	
Time step	MOP IDs	RTLs	MOP type IDs	Encoded fields	Inst. name	Format	Field values	Hardware cost	Inst. word width
1	mo1, mo2	m(R_1+D_1)<-R_2+D_2; R_3<-R_4+I	mrad, rrai	R_1=R_2=R_4; D_1=D_2=I	inst7	R_1, R_3, D_1	r2, r0, 0	1R, 1W, 1M, 1F	32
2	mo3, mo4	m(R_1+D_1)<-R_2+D_2; R_3<-R_4+I	mrad, rrai	R_1=R_2=R_4; D_1=D_2=I	inst7	R_1, R_3, D_1	r2, r1, 1	1R, 1W, 1M, 1F	32
3	mo6	pc<-pc+D	jd		inst5	D	1024	1F	22
4	mo5	R_1<-R_2+I	rrai		inst4	R_1, R_2, I	r2, r2, 2	1R, 1W, 1F	32

Table 4: Schedule II for the MOPs in Figure 1 and the resulted instructions

instruction format, i.e., the required instruction fields. The column 'Field values' lists the instantiated field values for the corresponding time step. Note that, in order to demonstrate the variation in instruction formation, the instruction set in Table 3 is chosen from a non-optimal one.

For example, in Table 3, the MOPs scheduled into time step 4 and 5 have the same binary tuple, and thus are mapped to the same instruction `inst4(R1,R2,I)`, with their field values instantiated to `(r1,r2,1)` and `(r2,r2,2)`, respectively. Note that we use capitalized letters, e.g. `R1`, to denote the instruction fields, and non-capitalized letters, e.g. `r2`, to denote the instantiated values of the fields. On the other hand, the MOP in time step 2, is mapped to a different instruction `inst2(R1,R2)`, although it contains the same type of MOP `rrai` as in time steps 4 and 5. Its field for the immediate data `I` is permanently assigned to the constant 'zero' and made implicit in the opcode, which is indicated by the specification `I=0`. This implicit field makes `inst2` behave as a 'move' instruction, instead of 'add'.

The compiled code can be obtained easily from the instruction names and instantiated field values. For example, the compiled code for the scheduled basic block in Table 4 is represented as the sequence: `inst7(r2,r0,0)`, `inst7(r2,r1,1)`, `inst5(1024)`, `inst4(r2,r2,2)`.

The instruction set is formed by unioning instructions generated from all time steps. For example, the instruction set derived from the schedule in Table 3 contains six instructions (`inst1~inst6`), and the instruction set for the schedule in Table 4 contains three instructions (`inst4,inst5,inst7`).

4.2. Performance and costs

The weighted sum of the lengths (number of time steps) of the scheduled basic blocks is the execution cycles of the benchmarks. The length of the basic block includes `nop` slots which are inserted by the design process to preserve the constraints due to multi-cycle operations. The design process will try to eliminate the `nop` slots by reordering other independent operations into the `nop` slots.

Each instruction has two costs associated with it. One is the total number of bits required to represent the instruction. The number is a summation of field widths of opcode and all explicit fields that are required to operate the MOPs contained in the instruction. The implicit fields do not consume instruction bits. For example, in Table 3, the instruction `inst4` requires 32 bits, using the bit width specification in Table 1; whereas `inst2` requires 16 bits only because its immediate data field is made implicit, saving 16 bits. The maximal bit widths of the instruction sets in Table 3 and Table 4 are 48 and 32 bits, respectively.

Another cost is hardware. It is the collection of the resources required by all MOPs contained in the instruction, minus the shared resources. The sharing of the resources can be related to field encoding. When two or more register reads of different MOPs are unified, i.e., reading from the same register, one read port of the register file is sufficient, instead of two or more. On the other hand, if more than one destination register receive results of the same arithmetic/logic expression, one functional unit is enough since the computation result can be shared. For example, `inst7` needs only one read port instead of three since `R1`, `R2`, and `R4` are unified. It also needs only one functional unit, instead of three, since the three destinations (memory data register, memory address register, register file) all receive the same value: `R1+D1`.

The global hardware resources are obtained by choosing the maximal number for each resource type from all instructions. For example, the global hardware resources used for the schedule I and II in Table 3 and Table 4 are <2R, 1W, 1M, 2F> and <1R, 1W, 1M, 1F>, respectively.

The example in Table 4 shows that compact and powerful instructions can be synthesized by packing more MOPs into a single instruction, and making fields implicit and register ports unified to satisfy the cost constraints. This is particularly useful in an application specific environment where instruction sets can be customized to produce compact and efficient codes for the intended applications.

4.3. Constraints

The MOPs are scheduled into time steps, subject to several constraints. First, the data/control dependencies and the timing constraints (for multi-cycle MOPs) have to be satisfied. Second, the instruction word width and the hardware resources consumed by the instructions have to be no larger than what are specified by the designer. Third, the size of the instruction set has to be no more than what the opcode field can afford.

4.4. Objective function

A richer instruction set may result in more compact and efficient compiled code, at the cost of complex decoding circuitry and design verification efforts. Therefore, an objective function is necessary to control the tradeoff of the cycle count *C* and instruction set size *S*, where *C* represents the performance metrics, how many cycles the benchmarks execute on the target machine, and *S* represents the cost metrics.

An interesting objective function suitable for our purpose is given by Holmer in [4]:

$$\text{Objective} = 100 \cdot \ln(C) + S \qquad \text{(EQ 1)}$$

Other types of objective functions can be used with the design system as well.

5. Simulated annealing algorithm

We use a simulated annealing scheme to solve the modified scheduling problem. An initial design state consisting of a schedule and its derived instruction set (generated by a pre-processor) is given to the design system, and then a simulated annealing process is invoked to modify the design state in order to optimize the objective function, until the design state achieves an equilibrium state.

The basic structure of our algorithm is similar to the simulated annealing algorithm for scheduling/allocation problem by Devadas and Newton [9]. Due to space limitation, we present only the move operators. For further information, please refer to [7].

The move operators change the design state. They provide methods of manipulating the MOPs and time steps. Currently the move operators implemented can be characterized into two groups. The first group manipulates the instruction format of a selected time step. There are five move operators in this group.

- *Generalization*: If the current instruction format of the selected time step contains encoded operands, make these operands general and become explicit in the instruction fields. The effects of this operator are increased instruction word width and hardware resources.
- *Unification*: Unify two register accesses in the MOPs; i.e., they always access the same register. For example, the specification of `R1=R2` in our previous example of the increment instruction `inc(R)` is a result of the 'unification' operator. The effects of this operator are the decreases in the instruction word width and/or register read/write ports.
- *Split*: Cancel the effect of the 'unification' operator. Two register accesses that are previously unified to the same register are made independent. The effects of this operator are the increases in the instruction word width and/or register read/write ports.
- *Implicit value*: Bind a register specifier to a specific register, or an immediate data field to a specific value. The specific values are the instantiated values in the MOPs of the selected time step. For example, the specification of `Immed=1` in the instruction `inc(R)` is a result of this operator. The effect of this operator is the decrease in the instruction word width.
- *Explicit value*: Cancel the effect of the 'implicit value' operator. Instruction fields that are previously bound to specific values are made explicit; i.e., their values are assigned by the compiler and are specified in the regular instruction fields. The effect of this operator is the increase in the instruction word width.

The second group of move operators involves the movement of the MOPs. There are four move operators in this group, which are all subject to the data/control dependencies and delay constraints when moving MOPs. The target MOPs and time steps can be selected randomly or with the guidance of heuristics.

- *Interchange*: Interchange the locations of two MOPs from different time steps.
- *Displacement*: Displace a MOP to another time step.
- *Insertion*: Insert an empty time step after or before the selected time step.
- *Deletion*: Delete the selected time step if it is an empty one.

These move operators have effects on performance (cycle count), hardware resources and instruction formation.

To improve the execution of the algorithm, there is a set of heuristics which selects proper move operators, based upon the current temperature and design status [7].

We have implemented the algorithm and its supporting tools into our design system ASIA (Automatic Synthesis of Instruction-set Architectures). It consists of about 5000 lines of Prolog code. In the following section we examine how the tools perform.

6. Experiments

In this section, experiments are presented to show the versatility of our tools by synthesizing instruction sets for some application benchmarks with various design constraints. Four benchmarks were selected from the Prolog Benchmark suite [12]. The benchmarks `con1` and `nreverse` are programs for list manipulation. The benchmark `query` is a program for database query. The benchmark `circuit` maps boolean equations into logic gates. The second column in Table 5 lists the characteristics of the benchmarks, including the numbers of MOPs, data-related dependencies, and control dependencies in the benchmarks. The number of MOPs represents the size of the benchmark; the number of data-related dependencies is related to the degree of parallelism available within the benchmark; the number of control dependencies indicates the degree of the impact of the branch/jump delays on the benchmark.

We assumed that every basic block executes once. We used the MOP specification as in Table 2. The delays are one cycle for branch/jump MOPs and zero for memory operation. The instruction field bit widths are given in Table 1. The experiment was conducted on a HP750 workstation with 256M memory.

For each benchmark, we synthesized its 32-bit, 48-bit, and 64-bit instruction sets[1], respectively. We were interested in how the instruction sets vary with bit widths. Table 5 lists the results, synthesized under the objective function with P=1 in (EQ 1). For all three benchmarks, as we had expected, the cycle decreases when the instruction word width increases. However, we observed a smaller gain in `nreverse`. This can be explained by its larger

1. The 48-bit and 64-bit versions of the benchmark `circuit` are not yet ready to be included in this manuscript.

Benchmark	# of MOPs, data dep., control dep.[*]	Instruction word width[†]	Design results				Performance of the algorithm	
			Cycle (C)	Instruction set size (S)	Instruction set space		Time (minutes)	Memory (megabytes)
con1	183, 136, 24	32	135	29	1275		56	2.1
		48	93	38	3733		59	2.7
		64	89	35	3277		48	2.7
nreverse	245, 395, 11	32	169	17	540		69	2.1
		48	157	23	772		57	2.0
		64	154	22	688		48	2.0
query	391, 185, 68	32	305	24	478		95	2.0
		48	215	32	1742		103	2.3
		64	204	39	1445		89	2.3
circuit	1725, 1077, 274	32	1406	40	1710		1358	3.4

Table 5: Results (Objective function = 100•ln(C)+S)

*. The number of control dependencies is counted as the total number of branch/jump micro-operations.
†. The hardware constraints are 3R, 1W, 2M, 1F for 32-bit instructions; 6R, 3W, 2M, 3F for 48-bit instructions; 8R, 4W, 32M, 4F for 64-bit instructions

ratio of the number of data dependencies to the number of MOPs. Most of the MOPs depend on each other such that there is less parallelism available when packing MOPs into instructions.

In general, the size of the instruction set also increases when the instruction word width increases. This is due to the fact that wider words can accommodate more MOPs, resulting in richer and more powerful instructions. However, the 48-bit instruction sets are ‘embarrassing’ designs for `con1` and `nreverse`. Their instruction set sizes are larger, but their performance is worse than their 64-bit alternatives in compiling the benchmarks. The 48 bits are not wide enough for these benchmarks to accommodate the most frequent MOP patterns, for which 64 bits are sufficient. Therefore, the design process has to specialize the general forms of some powerful instructions into several distinct instructions by making fields implicit or unifying register ports, in order to satisfy the bit width constraint.

In the ‘Instruction set space’ column we examined the number of instruction candidates explored by the design process. The numbers, much larger than the final instruction sets, show that the design process was able to explore a rich design space for the best candidates while keeping the size of the design space manageable.

In the two right most columns we also list the run time and memory usage of our algorithm, which show that our tools were able to synthesize instructions for application benchmarks within reasonable time and consume a modest amount of memory

In Table 6 we compared the synthesized 32-bit instruction sets with the BAM instruction set [13], which was designed for efficient execution of Prolog programs. The programs were compiled with the Aquarius Prolog Compiler, with the post-phase optimization phase turned off[2]. The experiments show that the synthesized instruction sets produced more compact codes for all four benchmarks, with 10%, 5%, 17%, and 3% reduction in the code size, respectively. This was achieved at the cost of a small number of additional instructions (7, 1, and 2 for `con1`, `nreverse`, and `query`, respectively), except in `circuit` where 16 additional instructions are required. We then used Holmer’s objective function (EQ 1) to evaluate the global performance/cost tradeoffs for both instruction sets and found that in most cases (`con1`, `nreverse`, and `query`) the synthesized ones yield better results, as indicated in the ‘Objective value’ column (smaller values are better). It is possible to improve the result of `circuit` by adjusting the initial temperature and the cooling schedule in our future experiment. We also compared the hardware resources used by both instruction sets. They both use the same amount of resources, except in the `nreverse` case our synthesized instruction set uses one less register read port and one less memory port than BAM does. This experiment shows that ASIA is capable of

2. The post-phase optimization of the Aquarius Prolog Compiler alters the classic definition of the basic block. Due to the time limit, we were not able to modify our tools to accommodate such change.

Benchmark	Instruction set	Hardware resources*	Cycle (C)	Instruction set size (S)	Objective value (smaller is better)
con1	BAM†	3R, 1W, 2M, 1F	150	22	523
	ASIA‡	3R, 1W, 2M, 1F	135	29	520
nreverse	BAM	3R, 1W, 2M, 1F	178	16	534
	ASIA	**2R**, 1W, **1M**, 1F	169	17	530
query	BAM	3R, 1W, 2M, 1F	368	22	613
	ASIA	3R, 1W, 2M, 1F	305	24	596
circuit	BAM	3R, 1W, 2M, 1F	1453	24	752
	ASIA	3R, 1W, 2M, 1F	1406	40	764

Table 6: Performance comparison with a manually designed instruction set

*. The resource constraints given to ASIA is the same as in the VLSI-BAM processor: 3R, 1W, 2M, 1F.
†. BAM refers to the instruction set that was manually designed for the VLSI-BAM processor.
‡. ASIA refers to the instruction set synthesized by the tools (ASIA) reported in this paper.

Instruction word width	RTLs*	Meaning
32	$m(R_1) \leftarrow R_2$; $R_1 \leftarrow R_1+D$	push†
	if (tf=1){ $m(R_1) \leftarrow R_2$; $R_1 \leftarrow R_1+D$}	conditional push†
	if (tag(R_1=T_1) { pc ← pc + D_1 }; if (tag(R_1=T_2) { pc ← pc + D_2 }	switch on tag†
48	tf ← R_1 OP. R_2; pc ← I	compute condition and jump
	$m(R_2) \leftarrow R_3$; if (tf=1){ $m(R_1) \leftarrow R_2$; $R_1 \leftarrow R_1+D$}	store and conditional push (with a shared register)
	$m(R_1) \leftarrow R_2$; $R_3 \leftarrow R_4+I$	store and add
64	$m(R_3) \leftarrow R_4$; if (tf=1){ $m(R_1) \leftarrow R_2$; $R_1 \leftarrow R_1+D$}	store and conditional push
	$m(R_1) \leftarrow R_2$; $R_3 \leftarrow R_4+ I$	store and add
	$R_1 \leftarrow T$ ^ I; $R_2 \leftarrow R_3 + I$	tag data and add

Table 7: Some synthesized instructions

*. Notations: 1). The RTLs in an instruction are executed simultaneously; 2). *tf*: a one bit latch which holds the truth value of a logic computation; 3). The operator ‘^’ appends a tag to a value before the value is sent to a destination.
†. These three instructions can be found in the BAM instruction set.

competing with manually designed instruction sets, based on the collection of the benchmarks. Further studies will be needed to investigate its competence in more general cases.

Table 7 shows some interesting instructions synthesized for the benchmark `query`. They are selected from the 32-bit, 48-bit, and 64-bit instruction sets, respectively. For ease of illustration, we do not list the binary tuples for these instructions; instead, we describe the RTLs of these instructions directly. In the RTLs, the register sharing is indicated by using the same register index. Note that the 32-bit version of the instructions can be found in the BAM instruction set as well. This fact provides the BAM designers with more confidence about their instruction set, since some of the instructions that they considered 'powerful' retain their existence when the instruction set is designed by other independent designers (in this case, the ASIA design automation system). This observation suggests that ASIA, in addition to its original purpose (an automatic design tool), can be used as a verification tool for designers to verify their manually designed instruction sets as well.

7. Conclusions

We have presented a design automation system ASIA (Automatic Synthesis of Instruction-set Architectures) that synthesizes computer instruction sets from application benchmarks. The design problem is formulated as a modified scheduling problem. The benchmarks are represented as data/control flow graphs of MOPs. The MOPs are scheduled into time steps subject to constraints of dependencies, hardware resources, and instruction word width. Instructions are formed during the scheduling phase. A binary tuple is used to describe the semantics and formats of instructions. The binary tuple is the key idea which links the instruction formation to the scheduling process. In addition to the synthesized instruction sets, ASIA also generates the compiled codes for the given benchmarks. An objective function of the cycle count and instruction set size is used to guide the design process, in order to balance the performance/cost tradeoff. A simulated annealing algorithm is used to solve for the schedules. We have discussed the move operators suitable for our problem.

We have demonstrated the versatility of ASIA by conducting experiments on some application benchmarks with various design constraints. The tools used reasonable amount of CPU time and a modest amount of memory. It has been shown that our tools are capable of synthesizing powerful instruction sets. Many of them can be found in today's processors. Compared with manually designed instruction sets, the synthesized instruction sets produce more compact code and may require less hardware. The tools were able to explore a rich design space, and handle important design options such as the instruction word width, and performance/cost tradeoff. We were able to explain the variation of the performance of the instruction sets on different benchmarks, based on the characteristics of the benchmarks. The experiments also show that ASIA, in addition to its original purpose in automating the design process, can be used by the designers to verify their manually designed instruction sets as well.

The current limitations include: First, the designers are required to specify the number of hardware resources, which may takes several iterations to find the best allocation. Second, in our problem formulation, the concept of the basic block is used to partition benchmarks into small pieces. However, there are other ways of partitioning benchmarks such as traces, and random segments [4]. What is the best way is unknown at this moment. Third, even though we have demonstrated that our algorithm is able to synthesize instruction sets from thousands of MOPs within 22 hours, real world application benchmarks, such as system, CAD and simulation software, are usually much larger. How to manage problems of such sizes is an important issue. Further researches need to investigate these issues.

Acknowledgment- The authors would like to thank Bruce Holmer, Ching-Long Su and the anonymous reviewers for their comments and suggestions in improving this work.

Reference

[1] F. M. Haney, "ISDS-A program that designs computer instruction sets," *Fall Joint Computer Conference*, 1969

[2] Pradip Bose and Edward S. Davidson, "Design of Instruction Set Architectures for Support of High-Level Languages," *Proc. of the 11th Annual International Symposium on Computer Architecture*, 1984

[3] J. P. Bennett, *A Methodology for Automated Design of Computer Instruction Sets*, Ph.D. thesis, University of Cambridge, Computer Laboratory, 1988.

[4] Bruce Holmer, *Automatic Design of Computer Instruction Sets*, Ph.D. thesis, Computer Science Department, University of California, Berkeley, 1993

[5] Jun Sato, et al., "An Integrated Design Environment for Application Specific Integrated Processor," *Proc. of ICCD*, 1991

[6] Ing-Jer Huang, Bruce Holmer and Alvin Despain, "ASIA: Automatic Synthesis of Instruction-set Architectures," *Proc. of SASIMI Workshop*, Nara, Japan, Oct. 1993

[7] Ing-Jer Huang and Alvin Despain, "Synthesis of Application Specific Instruction Sets," submitted to *Trans. on CAD*, 1994

[8] Peter M. Kogge, *The Architecture of Pipelined Computers*, McGraw-Hill Book Company, 1981

[9] Srinivas Devadas and Richard Newton, "Algorithms for Hardware Allocation in Data Path Synthesis," *IEEE Trans. on Computer-Aided Design*, Vol. 8, No. 7, July 1989

[10] Gert Goossens, et al., "An Efficient Microcode Compiler for Application Specific DSP Processors," *IEEE Trans. on Computer-Aided Design*, Vol. 9, No. 9, September 1990

[11] Shi-Zheng Lin, Cheng-Tsung Hwang and Yu-Chin Hsu, "Efficient Microcode Arrangement and Controller Synthesis for Application Specific Integrated Circuits," *Proc. of ICCAD*, 1991

[12] R. Haygood, *A Prolog Benchmark Suite for Aquarius*, Technical Report, UCB/CSD 89/509, University of California, Berkeley, 1989

[13] Bill Bush, et al., *The Berkeley Abstract Machine Instruction Manual*, Internal Technical Report, Advanced Computer Architecture Laboratory, University of Southern California, 1990

[14] Bruce Holmer, et al., "Fast Prolog with an Extended General Purpose Architecture," *Proc. of 27th International Symposium on Computer Architecture*, 1990