Generating Instruction Sets and Microarchitectures from Applications

Ing-Jer Huang and Alvin M. Despain
Department of Electrical Engineering – Systems
University of Southern California
Los Angeles, CA 90089-2561
ijhuang@usc.edu, despain@usc.edu

Abstract—The design of application-specific instruction set processor (ASIP) system includes at least three interdependent tasks: microarchitecture design, instruction set design, and instruction set mapping for the application. We present a method that unifies these three design problems with a single formulation: a modified scheduling/allocation problem with an integrated instruction formation process. Micro-operations (MOPs) representing the application are scheduled into time steps. Instructions are formed and hardware resources are allocated during the scheduling process. The assembly code for the given application is obtained automatically at the end of the scheduling process. This approach considers MOP parallelism, instruction field encoding, delay load/store/branch, conditional execution of MOPs and the retargetability to various architecture templates. Experiments are presented to show the power and limitation of our approach. Performance improvement over our previous work [4] is significant.

1. Introduction

Application specific instruction-set processors (ASIPs) offer a flexible and low cost solution for embedded systems with specific complex algorithms or control intensive applications [3][12]. As a result of the progress in design automation, it is now possible to synthesize ASIP-based embedded systems automatically. The synthesis of ASIP systems is an complex design problem which basically consists of three subproblems: microarchitecture design, instruction set design and instruction set mapping, which have been addressed by many people from different research areas.

High Level Synthesis (HLS), e.g., [7][9][10], generates microarchitectures at the RTL level from behavior specifications, which, in the context of ASIP design, are the instruction set specifications; Instruction Set Synthesis (ISS), e.g., [1][2][3][4], generates instruction sets from given descriptions of microarchitectures; Instruction Set Mapping (ISM), e.g., [13][17][18], compiles the given applications to assembly code, based on the given instruction set, so that the applications can be efficiently executed on the microarchitecture. Clearly, these problems are *not* independent ones. The need to closely examine the problems of HLS, ISS and ISM for instruction set processors has been noted by researchers, e.g., [2] and [12]. To investigate the interactions between these subproblems, most current approaches, e.g., [14][15][16], rely on manually controlled iterations between various synthesis tools.

This paper presents a method which expresses the synthesis of ASIP-based embedded systems, consisting of three subproblems HLS, ISS, and ISM, with a single formulation: a simultaneous scheduling/allocation problem with an integrated instruction formation process. As shown in Figure1, our method accepts applications expressed as dependency graphs of micro-operations (MOPs), an objective function, design constraints, and an architecture template, and generates the microarchitecture by allocating resources to the given architecture template, the application-specific instruction set, and the assembly code for the given applications. The design space of instruction sets consists of many features in modern pipelined processors, including parallel MOPs, operand (field) encoding, delay load/store/branch, and conditional execution of MOPs. The architecture templates that we consider are pipelined microarchitectures with a data stationary control model [6].

In this formulation, MOPs are scheduled into time steps, subject to instruction word width, dependency and timing constraints. While MOPs are scheduled into time steps, hardware resources are allocated, and instructions are formed at the same time. Note that instruction formation has effects on resource allocation as well. The assembly code is obtained from the schedule after the instruction set is finalized. The method has been implemented in our design automation system ASIA (Automatic Synthesis of Instruction-set Architecture).

The rest of the paper is organized as follows. Section 2 summarizes the basic problem formulation in our previous work [4] of which our current method is an extension. Section 3 describes the extension to accommodate microarchitecture design. Section 4 presents some experiments. Section 5 concludes this paper with discussions on the achievements, limitations, and future directions.

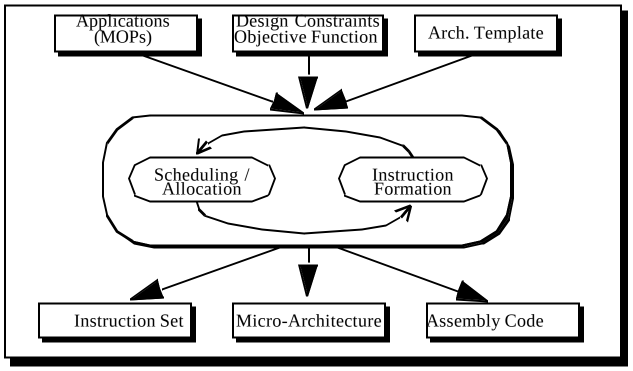

Figure 1. The scheduling/allocation process with an integrated instruction-formation process

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

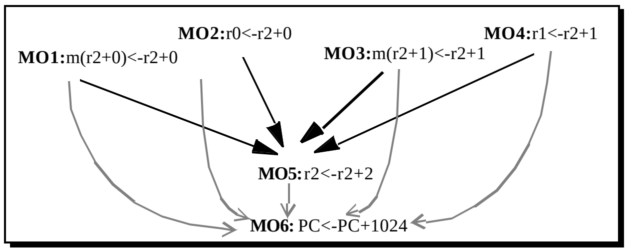

Figure 2. The dependency graph of MOPs of a simple basic block

2. Instruction set design and mapping as a scheduling problem

In this section we summarize the basic problem formulation of our previous work [4] which addresses the problems of ISS and ISM as a scheduling problem with an integrated instruction formation process. This formulation generates the instruction set and assembly code for the given application and microarchitecture. The extension to the basic formulation is given in Section 3 to include the synthesis of microarchitectures.

2.1. Representation of applications

An application is represented as a collection of weighted basic blocks. The weight is defined by the designer, and is usually used to indicate the typical execution frequency or the "importance", by some measure, of the basic block. The basic blocks are represented by dependency graphs of MOPs that are supported by the given microarchitecture. Figure2 shows an example of a basic block consisting 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; the MOPs MO6 changes the control flow at the end of the basic block, and hence logically follows MOPs MO1~5.

2.2. Instruction set model

An instruction contains one or more parallel MOPs. MOPs are controlled via instruction fields. The fields belong to some field types. The instruction word, consisting of fields, is assumed to be of fixed width. The widths of the instruction word and field types are specified by the designer. An example is given in Table1. 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)	8
immediate (I)	16
relation (<,=,>,≠) operator (OP)	2

Table 1: Bit width specification for 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; i.e., both register accesses refer to the same physical register) 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 parallel 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.

The semantics of an instruction can be represented by a binary tuple <*MOPTypeIDs*,*IMPFields*>, where *MOPTypeIDs* is a list of type IDs for the MOPs contained in the instruction, and *IMPFields* is a list of operand 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 [rrai]. Since no operand is encoded, the second argument of the tuple is an empty list. On the other hand, the binary tuple for the instruction inc(R) is <[rrai], [R1=R2,Immed=1]>. The list in the second argument of the tuple specifies how the operands 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.

2.3. Instruction formation and mapping

Instructions are formed while MOPs are scheduled into time steps by substituting the data values and register indices with general field templates. For example, the instruction add_store(R1,R2,R3,R4,Immed) 'R_1<-R_2+Immed; m(R_3)<-R_4' (';' represents parallelism) is formed when two MOPs 'r(15)<-r(11)+4' and 'm(r(11))<-r(11)' are scheduled into the same time step. If the design process decides to encode the instruction operands by unifying the register specifiers R_2=R_3=R_4, then this instruction becomes push(R1,R4,Immed) which is used in Prolog compilation [19]. This unification saves two register operand fields in the instruction.

The instruction set is derived from the final schedule and encoding decisions made by the scheduling process. The assembly code is obtained from the schedule and the synthesized instruction set.

2.4. Design constraints

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. If two dependent MOPs are required to be separated by certain cycles due to the timing constraint, independent MOPs, if available, or NOPs (no-operations) are filled in the delay cycles. 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 $2^{\text{opcode field width}}$.

2.5. A simulated annealing algorithm

We use a simulated annealing algorithm for the instruction formation and mapping problem. An initial schedule of the given application, produced by a preprocessor, is given to the algorithm as the initial design state. The algorithm then makes random movement in the design space by applying move operators to change the design state. In each movement, one MOP is selected randomly or according to some heuristics, and assigned to a randomly selected time step. Or, an instruction formation process can be applied to a time step to transform the semantics of the instruction at the time step. All of these movements are subject to the timing and dependency constraints.

The move operators which change the design state are grouped into two classes:

1. Manipulation of instruction semantics

   Unification: Unify two register operands in the MOPs. *Split*: Cancel the effect of the 'unification' operator. *Implicit value*: Bind a register operand field to a specific register, or an immediate data field to a specific value. *Explicit value*: Cancel the effect of the 'implicit value' operator. *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.

2. Manipulation of MOP's location

   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 and move one MOP to the new time step. *Deletion*: Delete the selected time step if it is an empty one.

In addition, there are some designer controllable parameters in the algorithm. The cooling schedule updates the current temperature. The movement accepting rules control the stability of design state at various temperature levels. The heuristics are used to select move operators and MOP targets when resolving violation of design constraints. For example, when the resource usage of a time step violates the resource constraint, the move operators unification, implicit value, interchange, displacement, insertion and deletion can be randomly selected to be applied to the time step.

3. Extension for microarchitecture design

The previous formulation can be extended to synthesize the microarchitecture at the same time, with the introduction of the architecture template specification language, resource allocation, a proper objective function, and design constraints which are discussed in this section.

3.1. Architecture templates

Ideally, it is desirable that the design automation system select the most feasible architecture template from a pool of candidate templates and carry out the design details. However, we have adopted a simpler approach for our current method, in order to manage the complexity of the problem. In our approach, the designer gives the architecture template by specifying the pipeline configuration and the general connection patterns of the data path. The algorithm then allocates appropriate hardware resources to instantiate the architecture template. The resources to be allocated are register read/write ports, memory ports, and functional units.

The styles of micro-architectures considered here are pipelined micro-architectures. The pipeline stages can be partitioned into two sections: the *instruction fetch* stages and the *instruction execution* stages. The instruction fetch stages are common to all pipeline configurations. The instruction execution stages are defined by the designer. For example, a basic pipeline, as shown in Figure3 (a), can be functionally partitioned into stages for instruction fetch (IF), instruction decode (ID), register read (R), arithmetic/logic operation (A), memory access (M), and register write (W). IF and ID belong to the instruction fetch stages, and R, A, M, and W belong to the instruction execution stages. Each functional stage may take more than one cycle, and can be further pipelined. The latencies of the functional stages are design parameters which are specified by the designer.

Other variations can be defined as well. For example, the pipeline 'IF-ID/R-A-M-W', the case (b) in the figure, can be derived by merging the register-read stage with the instruction-decode stage, at the cost of restricting the instructions to single format for register specification such that registers can always be prefetched at the instruction-decode stage. On the other hand, the pipeline 'IF-ID-R-A/M-W', the case (c), is derived by merging the arithmetic stage with the memory stage, at the cost of eliminating the displacement addressing mode. The displacements have to be computed by other instructions proceeding the memory-related instructions.

Figure4 (a) depicts the data path model of the pipeline in Figure3 (a). The register files at the top and bottom are the same register file. They are duplicated for the ease of readability. The data path model specifies the topology of modules, i.e., the connection patterns of data path modules. In the figure, there are paths R-A-M-W and its subpaths R-A-W, R-A-M, and M-W. There also exist paths A-R, M-R, and W-R which are created by the bypassing buses. The heavy dots in the figure indicate that the number of data path resources are to be instantiated by the algorithm.

The pipeline is controlled in a data stationary fashion [6]. In the data stationary control, the opcode flows through the pipeline in synchronization with the data being processed in the data path. Figure4 (b) shows the control path with data stationary model for the pipeline in Figure3 (a). Opcodes are forwarded to next stages

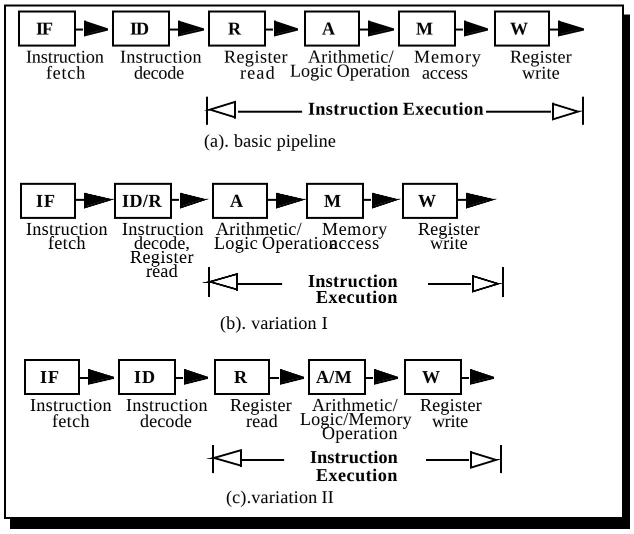

Figure 3. Basic pipeline and its variations

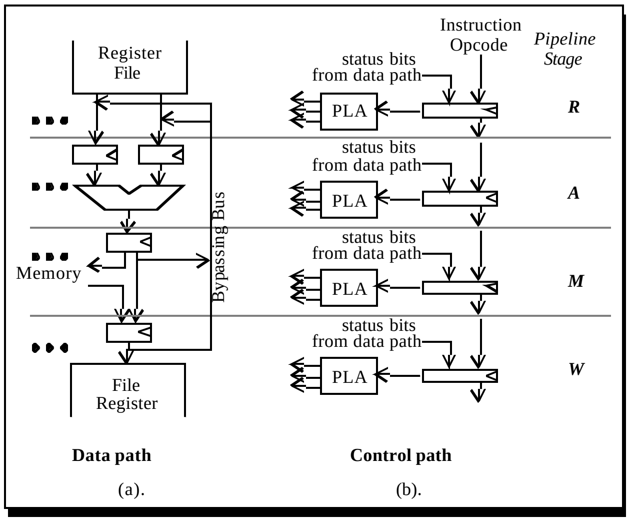

Figure 4. An architecture template

synchronously. At each stage, the opcode, together with possible status bits from the data path, is decoded to generate the control signals necessary to drive the data path.

3.2. The specification language for architecture templates

The architecture templates can be abstractly described by specifying the supported MOPs and a set of timing parameters. The MOPs describe the pipeline configuration of the instruction execution stages, the functionality supported by the microarchitecture, and the connectivity among modules in the data path. For example, in Table2 is the description of part of the MOPs supported by the architecture template in Figure4. Although not shown here, conditional execution of MOPs is allowed. Note that in this example, there is no direct connection between the register read and write ports. Therefore, a direct register move is not possible, which has to be achieved by some dummy arithmetic or logic operations such as adding with zero. The pipeline configuration 'IF-ID-R-A/M-W' in Figure3 (c) can be obtained by eliminating the MOPs rmd, mrd and mrad from Table2.

There are three attributes associated with each MOP. (1) The cost of the instruction format is the instruction fields required to operate the MOPs, including register specifiers, function selectors, and immediate data. (2) The hardware cost is the resources required to support the MOP. The hardware resources include read/write ports of the register file, memory ports, and functional units. (3) The required instruction execution stages are the pipeline stages in which the MOP is active. The third, fourth and fifth columns in Table2 lists these three attributes for the corresponding MOPs.

ID	MOP *	Inst. Format Cost†	Hardware Cost‡	Execution stages**
rra	R_1 <- R_1 + R_2	R_1, R_2	2 R, 1 W, 1 F	R, A, W
rrai	R_1 <- Immed + R_2	R_1, R_2, I	1 R, 1 W, 1 F	R, A, W
rrait	R_1 <- Tag^(Immed + R_2)	R_1, R_2, T, I	1 R, 1 W, 1 F	R, A, W
rmd	R_1 <- mem(R_2 + Immed)	R_1, R_2, I	1 R, 1 W, 1 M, 1 F	R, A, M, W
mr	mem(R_1) <- R_2	R_1, R_2	2 R, 1 M	R, M
mi	mem(R_1) <- Immed	R_1, I	1 R, 1 M	R, M
mrd	mem(R_1 + Disp) <- R_2	R_1, R_2, D	2 R, 1 M, 1 F	R, A, M
mrad	mem(R_1 + Disp) <- R_2+ Immed	R_1, R_2, D, I	2 R, 1 M, 2 F	R, A, M
jd	pc <- pc + Immed	I	1F	A

Table 2: MOP specification

*. The operator '^' appends a tag to a value before the value is sent to a destination.

†. Refer to the notation in Table1

‡. Notation: 'R'=read port of register-file, 'W'=write port of register-file, 'M'=memory port, 'F'=functional unit, and the value is the number of a particular hardware resource. For example, '2R' means two read ports for register-file.

**. Notation: 'R'=register read stage; 'A' =ALU stage, 'M'=memory stage, 'W'=register write stage

The set of timing parameters describes the operation latencies in terms of clock cycles. There are two classes of operation latencies: latencies for the operations of data path modules, and latencies (delay cycles) for information passing between operations in pipeline stages. Examples for the architecture template in Figure4 are shown in Table3 and Table4, respectively. The M-A pair in Table4 specifies that there should be one cycle delay between a memory operation and a succeeding (dependent) arith-

Data path module	Latency
Register read	1
Register write	1
Memory access	2
Arithmetic/Logic	1

Table 3: Timing parameters: latencies of data path modules

Operation pair	Delay cycles	Operation pair	Delay cycles
arithmetic-arithmetic (A-A)	0	memory-control (M-C)	1
arithmetic-memory (A-M)	0	control-arithmetic (C-A)	1
arithmetic-control (A-C)	0	control-memory (C-M)	1
memory-arithmetic (M-A)	1	control-control (C-C)	1
memory-memory (M-M)	1		

Table 4: Timing parameters: latencies of operation pairs

Note that the design parameters are able to model the existence of bypassing buses in the data path. For example, if we remove the bypassing bus in the 'A' stage in Figure4, then the delay cycles for the A-A, A-M, and A-C pairs in Table4 all become one, instead of zero.

3.3. Resource allocation

Hardware resources are allocated while MOPs are scheduled to time steps. This is similar to the problem formulation in [8]. For each fortime step, the required hardware resources are the total of the resources consumed by each MOP scheduled into the time step, minus the resources that are shared. The sharing of resources in a time step is due to the operand encoding. When two or more register reads belonging to different MOPs are unified, i.e., reading from the same register, one register read port is sufficient. 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.

Take the instructions discussed in Section 2.3 for example. The instruction add_store(R1,R2,R3,R4,Immed) requires *three* register read ports for the register read specifiers R2, R3 and R4, one register write port for R1, one functional unit for addition, and one memory port for the store operation. On the other hand, the instruction push(R1,R4,Immed) requires only *one* register read port, as opposed to three, with the requirements for other types of resources remaining unchanged. The saving is due to the unification of register read specifiers R_2=R_3=R_4.

The global resources allocated is then the union of the resources used by each instruction.

3.4. Design constraints and objective function

The design constraints used in Section 2 remain intact, except the resource ones which are eliminated from the problem formulation. The algorithm is responsible for finding the best resource allocation according to the objective function.

The goal of the algorithm is to minimize the value of the objective function which is given by the designer. The objective function can be an arbitrary function of the dynamic cycle count *C*, the static code size *S*, the instruction set size *I*, the number of register read (write) ports *R* (*W*), the number of memory ports *M*, and the number of functional units *F*.

3.5. Design process

The design process for the HLS+ISS+ISM problem consists of three phases.

1. The given application is translated to dependency graphs of MOPs which are supported by the given architecture template. This translation is performed in two steps. First, the application, written in a high level language, is translated into an intermediate representation by the compiler of the high level language (in our current environment, the Aquarius Prolog Compiler [19]). Second, a retargetable MOP mapper, consulting the given architecture template specified with the language described in Section 3.2, transforms the intermediate representation into the dependency graphs of MOPs.
2. A preprocessor generates a simple-minded schedule for the MOPs. An instruction set is derived from the schedule. This is done by directly mapping time steps in the schedule into instructions without encoding any operand. The obtained schedule and instruction set constitute the initial design state, which can be an inferior one.
3. The simulated annealing algorithm, with the modifications discussed in Section 3.3 and Section 3.4, is invoked to optimize the design state. Note that the initial temperature for the annealing process has to be higher than the problem in Section 2. The number of movements tried at each temperature has to be larger as well. These modifications are due to the much larger design space when instruction sets and microarchitectures are designed together. Several experiments may be necessary in order to set the proper values for these parameters.

The best instruction set, microarchitecture, and assembly code which minimize the objective function can be obtained after the design state reaches the equilibrium state.

4. Experiments

In this section we present experimental results of our simulated annealing algorithm for the design of application specific instruction sets and microarchitectures, and instruction set mapping. To simplify the experiments, an application specific design was synthesized for each given application. However, this is not to mean that our algorithm can only be used to synthesize designs for single applications. Application specific designs customized for a set of given applications can be synthesized by taking the collection of the MOP dependency graphs from all the given applications as the input of the algorithm.

We used the MOP specification in Table2 and timing parameters in Table3 and Table4 as the given architecture template. The bit width constraints for instruction fields is given in Table1. The following function is used as the objective function. The meanings of the variables are given in Section 3.4.

$$\textit{Objective} = 100\ln(C) + I + 2R + 3W + 5M + 4F \qquad \text{EQ 1}$$

We assumed that every basic block executes once for every application. This assumption was due to the fact that the profile analyzer was not available at the moment so that we were not able to obtain the run time behavior, e.e., the execution counts of basic blocks. The number of movements tried at each temperature point is *5*(# of MOPs)*. The next temperature is 90% of the current temperature. The experiments were conducted on a HP 750 workstation with 224M memory.

Three symbolic applications were selected from the Prolog benchmark suite [11]. *hanoi_8* is the 'hanoi' problem solver. *con1* concatenates two strings into one string. *nreverse* reverses the order of the given string. Note that the predicate concat defined in *con1* is used as a subroutine in *nreverse* as well. Figure5 lists the source code in Prolog. The main clauses are given by the designer to represent the typical execution of the programs.

```
main :- hanoi(8).
hanoi(N) :- move(N,a,c,b).
move(0,_,_,_) :- !.
move(N,A,B,C) :- M is N-1,
        move(M,A,C,B), move(M,C,B,A).
```

(a). hanoi_8

```
main :- concat([a,b,c],[d,e],_).
concat([],L,L).
concat([X|L1],L2,[X|L3]) :-
        concat(L1,L2,L3).
```

(b). con1

```
main :-
    nreverse([1,2,3,4,5,6,7,8,9,10,11,12,
        13,14,15,16,17,18,19,20,21,
        22,23,24,25,26,27,28,29,30],_).
nreverse([X|L0],L) :- nreverse(L0,L1),
    concat(L1,[X],L).
nreverse([],[]).
concat([],L,L).
concat([X|L1],L2,[X|L3]) :-
    concat(L1,L2,L3).
```

(c). nreverse

Figure 5. Application programs in Prolog

The results are given in Table5. The columns under the header "Synthesis results" are the outputs of the algorithm: the resource allocation, cycle counts of the application, the instruction set size. Note that *con1* and *nreverse* have more MOP parallelisms available than *hanoi_8*. Therefore, *con1* and *nreverse* are allocated more hardware resources than *hanoi_8*. Furthermore, some powerful instructions are included in *con1*'s and *nreverse*'s instruction sets to make use of the additional hardware resources, as indicated by the larger sizes of their instruction sets. In the table we also list the number of candidate instructions (in the "Inst. set space" column) and the number of microarchitecture configurations (in the "uArch space" column) explored by the algorithm. These numbers show that sufficient number of design candidates have been explored.

Benchmark	Synthesis results				
	Resource allocation*	Cycle (C)	Inst. set size (S)	Inst. set space	uArch space
hanoi_8	2R, 1W, 1M, 1F	38	19	126	8
con1	3R, 1W, 1M, 1F	135	24	244	7
nreverse	3R, 1W, 2M, 2F	313	25	275	7

Table 5: Synthesis results

*. Refer to the footnote '‡' in Table2 for the meanings of the notation.

For comparison, we also applied an iterative approach with our previous tool [4] to find the best microarchitectures and instruction sets for the applications. Our previous tool was constructed to generate instruction sets and mapping for given resource allocation. Therefore, the most difficult task is to decide that how many configurations of resource allocation should be tried before we believe that sufficient design space has been explored. For each application, we chose three resource configurations from the "uArch space" column in Table5. The three configurations include the best one (in the "Resource allocation column in Table5) and two other possible configurations that are closest to the best one[1]. Table6 lists the objective values and the CPU times used by the tools. The header "The integrated approach" lists the results of the algorithm presented in this paper. The header "The iterative approach" lists the results of our previous tool. The results show that the integrated approach finds the same or better designs than the iterative approach. And the integrated approach has an average speedup of 3.5 in CPU time over the iterative approach. This is because that while the iterative approach has to conduct several complete runs with various resource configurations in order to find the best solution, the integrated approach find the best solution much faster by dynamically switching between different resource configurations so that infea-

1. In real designs, more possible configurations have to be tried, in order to ensure that sufficient design space has been explored.

sible design space can be pruned early in the search process. The comparison shows that the integrated approach, in addition to the clarity in the problem formulation, is a significant performance improvement over our previous approach in solving the combined HLS+ISS+ISM problem.

It is difficult to fairly compare our approach with related work since these approaches evolve from different research disciplines such as computer architecture, compiler and high level synthesis. They have different concerns, machine models and problem formulations. Further investigations are necessary to compare related work in the future.

Benchmark	The integrated approach		The iterative (previous) approach		Speedup (T_2/T_1)
	Objective value	Time T_1 (min.)	Objective value	Time T_2 (min.)	
hanoi_8	398.76	10	398.76	38	3.80
con1	532.52	44	535.58	147	3.34
nreverse	626.62	197	634.30	682	3.47
				Average	3.54

Table 6: Comparison with our previous approach [4]

Due to space limit, we do not list the synthesized instruction sets in the paper. Interested readers may refer to [5]. By carefully examining the synthesized instructions, we found that there exist few powerful instructions that are rarely used in the assembly code. The instruction set sizes can be reduced by deleting these instructions, at the cost of slightly increased cycle counts. This action will further reduce the objective values, resulting in better solutions. However, this was not done by our algorithm. The reason is that the chance of the MOPs contained in these instructions being selected and displaced by the algorithm was very low since this pattern occurred rarely in the application. To fix this problem, we can increase the number of movements tried at each temperature point at the cost of increased CPU time. Or, we can introduce more powerful move operators such as “delete an instruction” or “delete a resource” to the algorithm at the cost of complicating the design heuristics and modification to the data structure, since the objects being moved is no longer just MOPs, which are simple and local, but also instructions and resources, which are complex and global.

The experiments also demonstrate that our tools can be used for the exploration and analysis of several interesting architectural properties of applications. However, the space limit does not allow the discussion. Interested readers may refer to [5].

5. Conclusions

We have presented a method which encapsulates the combined problem of instruction set design, microarchitecture design and instruction set mapping with a single formulation: a simultaneous scheduling/allocation problem with an integrated instruction formation process. The formulation takes as inputs the application, architecture template, objective function and design constraints, and generates as outputs the instruction set, resource allocation (which instantiates the architecture template) and assembly code for the application. A simulated algorithm is used to solve the problem. The method is an extension to our previous work [4].

We have presented experimental results with three Prolog applications. Limitations of the method are shown, and possible improvements are discussed. The experiments also show that the current method has significant speedup over our previous method coupled with an iterative approach.

In the future, we need to address the following problems. (1) The CPU time grows quickly with the size of application. For larger applications, the design may be accomplished in two phases. In the first phase, the integrated synthesis task is performed on the most important part of the applications. In the second phase, instruction set mapping is performed for the rest of the applications, based on the design derived on the first phase. (2) The automatic generation of simulators for the synthesized instruction sets and microarchitectures is necessary for the purposes of verification and performance measurement. (3) Similar to [8], binding and connection synthesis can be integrated into the design process as well.

Reference

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

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

[3] Alauddin Alomary, et al., “An ASIP Instruction Set Optimization Algorithm with Functional Module Sharing Constraint,” *Proc. of the International Conference on Computer-Aided Designs*, Nov. 1993

[4] Ing-Jer Huang and Alvin Despain, “Synthesis of Instruction Sets for Pipelined Microprocessors,” *Proc. of the 31st Design Automation Conference*, June 1994

[5] Ing-Jer Huang, *Co-Synthesis of Instruction Sets and Microarchitectures,* Ph.D. thesis, Dept. of Electrical Engineering - Systems, Univ. of Southern California, August 1994

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

[7] Mauricio Breternitz Jr. and John Paul Shen, “Architecture Synthesis of High-Performance Application-Specific Processors”, *Proc. Design Automation Conference,* 1990

[8] 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

[9] Ing-Jer Huang and Alvin Despain, “Hardware/Software Resolution of Pipeline Hazards in Instruction Set Processors,” *Proc. of the International Conference on Computer-Aided Designs*, Nov. 1993

[10] Richard Cloutier and Donald Thomas, “Synthesis of Pipelined Instruction Set Processors,” *Proc. of 30th DAC*, 1993

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

[12] Pierre Paulin, Clifford Liem, Trevor May, Shailesh Sutarwala, “DSP Design Tool Requirements for Embedded Systems: A Telecommunications Industrial Perspective,” to appear in *Journal of VLSI Signal Processing*, 1994

[13] Clifford Liem, Trevor May, Pierre Paulin, “Instruction-Set Matching and Selection for DSP and ASIP Code Generation,” *Proc. of EDAC*, 1994

[14] Johan Van Praet, Gert Goossens, Dirk Lanneer, Hugo De Man, “Instruction Set Definition and Instruction Selection for ASIPs,” *Proc. of Int’ l Symposium on High Level Synthesis*, May 1994

[15] Bruce Holmer and Barry Pangrle, “Hardware/Software Codesign Using Automated Instruction Set Design & Processor Synthesis,” *Proc. of Hardware/Software Codesign Workshop*, 1993

[16] Hironobu Kitabatake and Katsuhiko Shirai, “Functional Design of a Special Purpose Processor Based on High Level Specification Description,” *IEICE Trans. Fundamentals*, Vol. E75-A, No. 10, Oct. 1992

[17] M. Corazao, et al., “Instruction Set Mapping for Performance Optimization,” *Proc. of ICCAD*, Nov. 1993

[18] Wei-Kai Cheng and Youn-Long Lin, “Code Generation for a DSP Processor,” *Proc. of Int’ l Symposium on High Level Synthesis* , May 1994

[19] Peter Van Roy and Alvin Despain, “HIgh-performance Logic Programming with the Aquarius Prolog Compiler,” *Computer*, 25(1):54-68, January 1992