

Automatic Assembly Program Retargeting for Microcontrollers

ING-JER HUANG AND DAO-ZHEN CHEN
Department of Computer Science and Engineering
National Sun Yat-Sen University
Kaohsiung, 804 Taiwan
E-mail: ijhuang@cse.nsysu.edu.tw

A new approach is proposed to translate existing software programs from one instruction set other instruction sets at the assembly level. The behaviors of instructions are abstractly represented as manipulation of the machine state. The behavior of each basic block of the software program is then represented as state transition pair. Instruction set retargeting is then modeled as the problem of finding sequences of instructions accomplishing the same machine state transitions at the target machine as does the software program at the source machine. The proposed approach has been successfully demonstrated for software translation between several industrial microcontrollers.

Keywords: retargeting, compiler, computer architecture, state abstraction, assembly programs

1. INTRODUCTION

The fast progress in IC design technologies has produced many new microcontrollers/microprocessors with better code density, more functionality and lower power consumption for portable and consumer electronics products, where the issue of cost/functionality is more important than the issue of software compatibility. To replace old microcontrollers with new ones in products often leads to a challenge: in order to preserve the original software investment and to shorten the time-to-market, the existing software that has been developed under the old microcontrollers has to be retargeted (ported) to the new microcontrollers (if they have different instruction sets, which is usually true if better code density or more functionality is desired). However, unlike the environment of desktop computers, a significant portion of the software is usually developed directly at the assembly or binary level to optimize the cost or performance of these products. In such cases traditional retargetable compilers based on high level languages are of little help.

In this paper, we present a new approach to the instruction set retargeting problem at the assembly level based on the *machine state transition* notation. This is motivated by the observation that the goal of retargeting is actually to look for an instruction sequence of the new instruction set that produces (emulates) the *same machine state* as the instruction sequence of the original instruction set. As long as the same machine state is reached,

it does not matter whether the two instruction sequences come from the same grammatical expressions (or trees, graphs, *etc.*) or not. This observation is especially applicable to microcontroller-based embedded systems since the major purpose of the software running on them is to monitor and control the machine state (status). What programmers care about most is how the machine state transits rather than the operations of the instruction set itself.

The machine state based retargeting approach has been successfully applied to solve a special version of the retargeting problem: constructing the x86-to-RISC instruction mapping table for an embedded RISC processor core [1]. The embedded core can be considered as an application specific instruction set processor (ASIP) whose sole application is to efficiently emulate x86 instructions. An x86 instruction is fetched into the core and then decoded on the fly into simpler RISC instructions according to the mapping table for efficient execution. In this case, retargeting is performed instruction by instruction.

The work presented in this paper is an extension of the above work that expands the translation scope from individual instructions to entire software programs for microcontrollers. This extension makes it possible to perform instruction retargeting, code optimization and verification with the same machine state notation.

The rest of the paper is organized as follows. Section 2 reviews related work. Section 3 discusses the problem modeling. Section 4 presents the retargeting framework. Section 5 demonstrates the proposed technique with translation between three industrial 8-bit microcontrollers: PIC (RISC-like), MC8052 (CISC-like) and HT48100 (RISC-like).

2. RELATED WORK

Most of the existing retargeting compiler systems are based on graph or tree matching algorithms. Aho *et al.* [2] proposed an optimal dynamic-programming-based tree matching algorithm for retargeting compilers. Marwedel also presented a tree-based approach for mapping to a predefined hardware structure [4]. Corazao [6] proposed a matching method for DSP processors, based on templates of the CDFG's (control/data flow graphs) of instructions. Liem *et al.* [7] used rule-driven compilation. It has a shorter compilation time than pattern matching does. An extensive review of retargetable code generation theories and practices can be found in the book by Marwedel and Goossens [8].

These matching algorithms usually need high level source code in order to generate the necessary trees or graphs for matching. Therefore, they are not well suited for binary or assembly level retargeting because source code is not available.

Sites *et al.* [5] developed a binary translator that translates the VAX VMS and MIPS Ultrix binary code into DEC Alpha AXP and its execution environment. They built a translator called VEST and a run time environment called TIE. The translator maps the VAX code to AXP code according to a mapping table. When the translator encounters the portion of the VAX code that the translator is unable to identify, that is, whether it is the program or the data, the portion is embedded in the new AXP code. The VAX code embedded in the AXP code is executed by a run-time interpreter. Another similar work is Digital's FX!32 [3]. We are interested in how the mapping tables are constructed, but unfortunately, that was not discussed in the above papers. We guess that they were con-

structed manually since their target platforms were fixed and, therefore, a one-time, ad-hoc effort was sufficient.

C. Cifuentes cooperated with N. Ramsey to develop an integrated reverse engineering environment for binary code which is capable of translating binary programs from a given machine to a different machine [9]. The binary translation is achieved in three phases: a front-end to translate the binary representation into an intermediate representation, a middle-end to perform analysis and optimization, and a back-end to map the intermediate representation to the binary representation of the target machine. The retargeting tool is built upon a syntax specification language SLED [10] and a semantics specification language SSL [11]. Our work is different from theirs in two respects. First, their techniques are targeted on general purpose microprocessors, such as SPARC, x86 and PowerPC, while ours is targeted on application specific embedded systems. Second, we do not rely on the typical intermediate representation, found in many retargeting research works, as the interface between two instruction sets. Instead, we abstract the source instructions into machine state transitions and then try to accomplish the same machine state transitions with the target machine instructions.

In our previous work [1], we proposed a state notation to guide instruction-to-instruction retargeting, but we have not considered dependency schedule issues. In addition the phase that performs state abstraction of a basic block has not yet been described. In this paper, we consider the retargeting of basic block, and extend this idea to application program retargeting.

3. PROBLEM MODELING

3.1 Machine States

3.1.1 State abstraction

A more effective way to study the behaviors of instructions is to observe their effects on the storage elements, such as registers, flags, latches and memory, which together define the characteristics or the state of the entire machine. The right side of Fig. 1 shows the machine state of the microarchitecture, shown on the left side of Fig. 1. The ultimate objective of instructions is to manipulate the machine state; the operations (of the instructions) in the microarchitecture are just the means used to accomplish such manipulation. Therefore, the machine state can serve as an abstraction for the machine. This state-based abstraction is more suitable for the instruction retargeting problem since observing how the machine state is modified by instructions requires less information and effort than does observing the data path structure and detailed operations in the microarchitecture. In addition, the machine state viewpoint makes it easy to accommodate variations in the microarchitecture and the instruction set: two pieces of software code, although different in their contents or even in the instruction sets that they are based upon, can be regarded as compatible as long as they carry out the same state transition (from the same initial state to the same final state).

Based upon the above concept, we construct the machine *state* with a list of symbolic values of storage locations, called *contents*. The content of each storage location can be expressed as a binary tuple: $\text{content}(\text{Location}, \text{Value})$, where *Value* is the symbolic

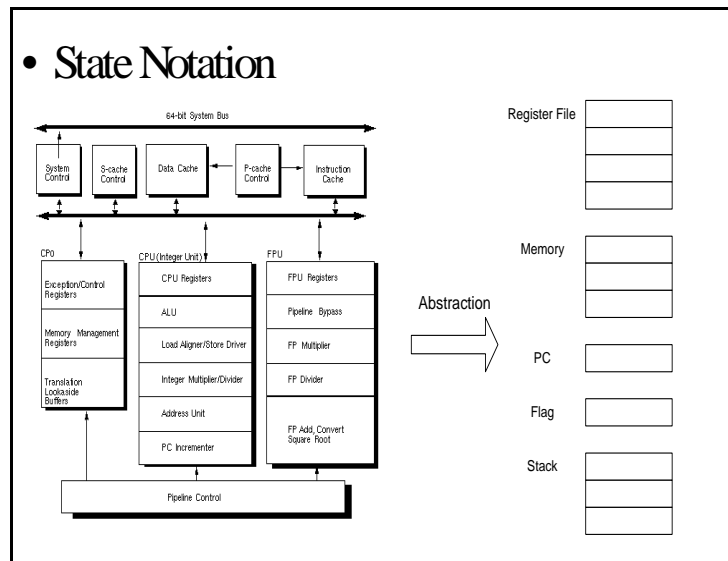


Fig. 1. State abstraction concept.

value of the storage element in Location. The storage location can be a special or general register, a memory word, a latch, an IO port, *etc.* The symbolic value can be a constant, a value from another location, or an expression comprising constants and values from some storage locations. For example, the instruction `add a, b, 1` ($a = b + 1$) is represented as `content(reg(a), reg(b) + immed(1))`, which shows that the register location `reg(a)` gets the value of the register `reg(b)` plus the immediate value of one. Notice that the register index can be physical or symbolic. In the latter case, register allocation is necessary to couple the retargeting process. Since a machine state may consist of numerous storage locations (e.g., a few giga-words of memory), only the locations that are modified or that are of particular interest need to be explicitly specified.

The above binary tuple needs to be extended to support conditional states. In this case, the machine state is represented by the triple `content(Condition, TrueStateList, FalseStateList)`, where `Condition` is a Boolean expression, and `TrueStateList` and `FalseStateList` are the machine state (i.e., a list of contents as defined previously) under the true and false conditions, respectively.

3.1.2 State representation of basic blocks

The machine state representation of a basic block is derived by consolidating the machine states of individual instructions in the basic block. To consolidate the machine states, we take the union of the machine states of individual instructions and perform constant/expression propagation if some instructions reuse some parts of the machine state. For example, suppose a basic block has a simple instruction sequence `[mov a, b; mov c, a]`. After processing the first instruction, the machine state becomes `[content(reg(a), reg(b))]`. After processing the second instruction, the machine state becomes `[content(reg(a),`

tent(reg(a), reg(b)), content(reg(c), reg(b))]. Note that the original semantics of the second instruction copy the value of register a to c. However, since the initial value of register a is stored with the initial value of register b by the first instruction, the final value of register c is represented as reg(b) instead of reg(a).

One of the advantages of the machine state representation is that while abstracting the basic block behavior into states, some code optimizations can be automatically (implicitly) achieved. Some examples are given below.

Eliminate architecture bias

Fig. 2 shows the state representation of a simple basic block containing two instructions. The instructions are part of the instruction set of a two-operand architecture. The derived state specifies that the effect of the basic block is that register a to get the summation of registers b and c. Note that in the state representation, the effect of the two-operand architecture disappears. When the basic block is translated into a three-operand architecture according to the state representation, it will be directly translated into one instruction, such as the add a, b, c instruction. In this example, the architecture dependent feature (bias) is automatically eliminated without extra effort.

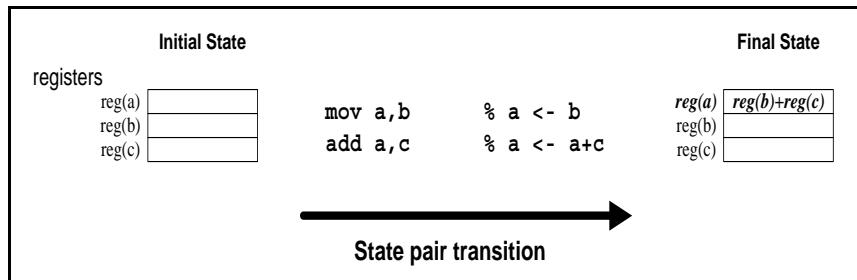


Fig. 2. Implicit optimization with state abstraction (1).

Produce less dependent code

Consider the basic block shown in Fig. 3, which copies the value of register b to register a first and then copies the value of register a to c. There is data dependency on register a. However, when we abstract the basic block, the net effect is that the value of register b is duplicated in both register a and c. The data dependency is implicitly eliminated by the state abstraction process.

Produce less redundant code

Redundant code can also be automatically removed by the state abstraction process. For example, the second instruction of assembly code shown in Fig. 4 is a redundant instruction that does not create any new machine state. The corresponding state representation is simply [content(a, b)], as shown in the figure. When this basic block is mapped to other instruction sets based on the derived state representation, only one instruction will be needed, instead of two instructions as in the original inefficient code.

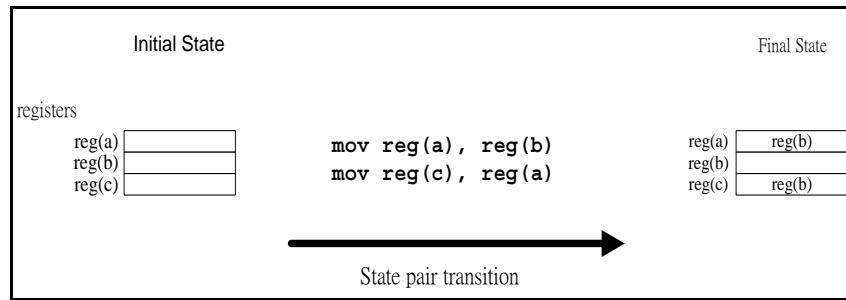


Fig. 3. Implicit optimization with state abstraction (2).

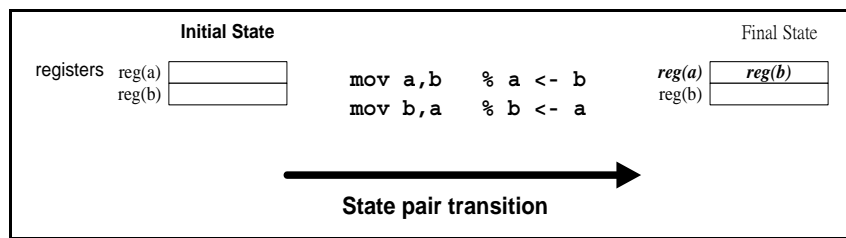


Fig. 4. Implicit optimization with state abstraction (3).

3.1.3 Special microarchitecture features

3.1.3.1 IO instruction manipulation

When we make the abstraction of basic block [mov reg(r1), 10; mov reg(r1), 20; mov reg(r1), 30], we will have the result as reg(r1) will get the final state as 30, but if we consider the state abstraction of basic block [mov port(p1), 10; mov port(p1), 20; mov port(p1), 30], then port(p1) is the output port, and the situation is much different. From our viewpoint of system state abstraction, the final state of port(p1) is 30, but a port is not considered in the same manner as the internal system state (for example, the register). Therefore, the system state change of port during basic block execution is important for the external environment because the port is accessible by an external system. Therefore, in the previous example, we should divide the three out port instruction into three consecutive sub-basic-blocks.

Fig. 5 shows the method we use to deal with this situation. The three output port instructions are divided into three consecutive basic blocks. This is because they have the same port. However, it is very rare that data is outputted to the same port more than once in a basic block.

3.1.3.2 Side effect modeling

Some instructions will change the system flag in addition to the normal operand as side effects of the operation. In this case, the change of flag is modeled as a side effect in

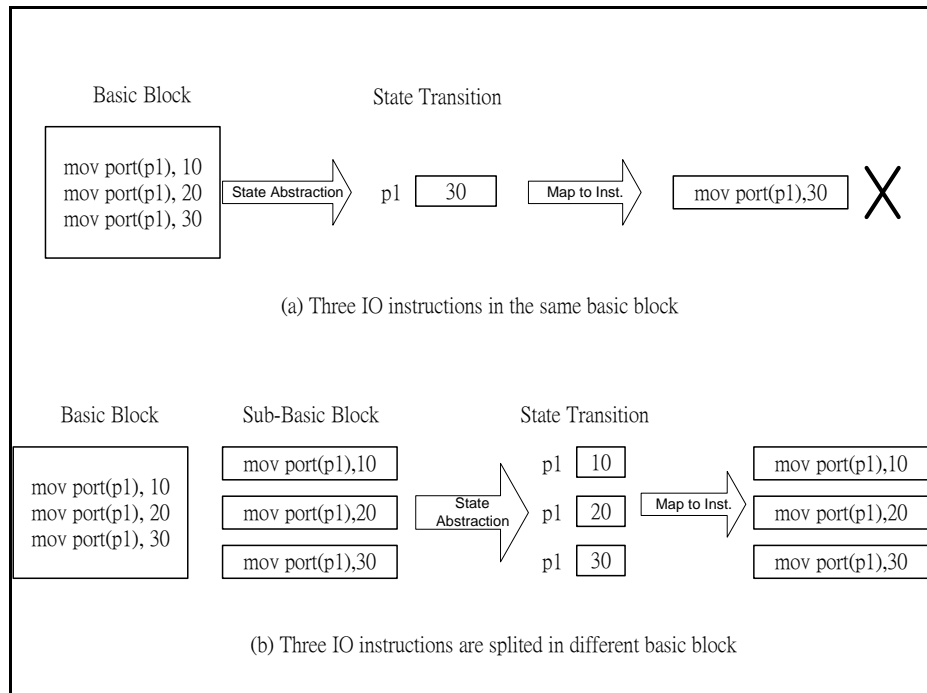


Fig. 5. Split I/O instructions among different sub basic blocks.

our work. Because program control flow (such as a conditional jump) always depend on some flag conditions, it is very important to include the flag matching algorithm in our retargeting engine. For example, when **add** instruction is executed, the **c** (carry), **ac** (auxiliary carry), **ov** (overflow), and **z** (zero) **flags** will change according to the result of the operation. In this situation, our mapping rule is augmented with the side effect mapping rule. Fig. 6 shows the state transition of three instructions with side effect modeling taken into account. The **mov** instruction does not modify any flag, the **add** instruction **c**, **ac**, **ov**, and **z** **flags**, and the **and** instruction modifies the **z** flag. Fig. 7 shows that after the three instructions are executed, the flag will record the result as **c**, **ac**, and **ov** **flags** that will be modified according to the result of the **add** instruction, while the **z** **flag** will be modified according to the result of the **and** instruction. The state abstraction algorithm of the side effect is implemented as **side effects caused by former instructions that are overwritten by side effects caused by later instructions**. Therefore, the side effect setting of the **z** **flag** set by the **add** instruction will be overwritten by an **and** instruction.

When side effect modeling is taken into consideration, the mapping rule should also be modified to match the side effect. The algorithm should also match the side effect while deciding which operator can be used to match the source state transition. For example, as Fig. 8 shows, the algorithm will choose **and d, e** instead of **add a, b, c** due to consideration of the side effect. If **add a, b, c** is chosen first, we can see that the **z** **flag** will not be matched because the **z** **flag** can be matched only by the ‘&’ operation. If **and d, e** is chosen first, we can see that the operation and side effect can be perfectly matched; then, the **z** **flag** is unified as “_”, which means **don’t care**. After that, when **add a, b, c**, is chosen next, the **z** **flag** can be (unified) matched with any operation (because it is modified as don’t care) and thus can be matched.

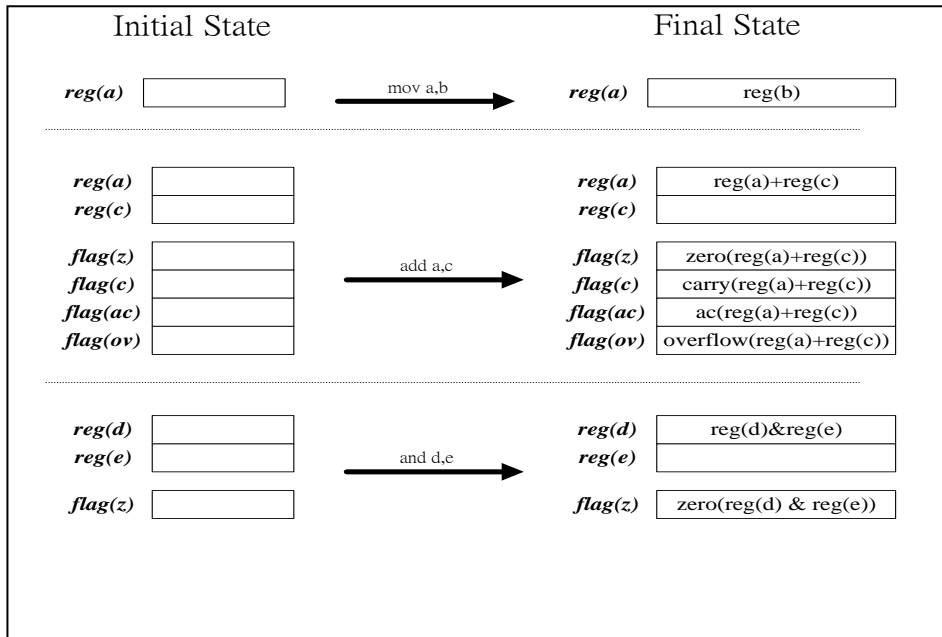


Fig. 6. State abstraction of an instruction (with the side effect).

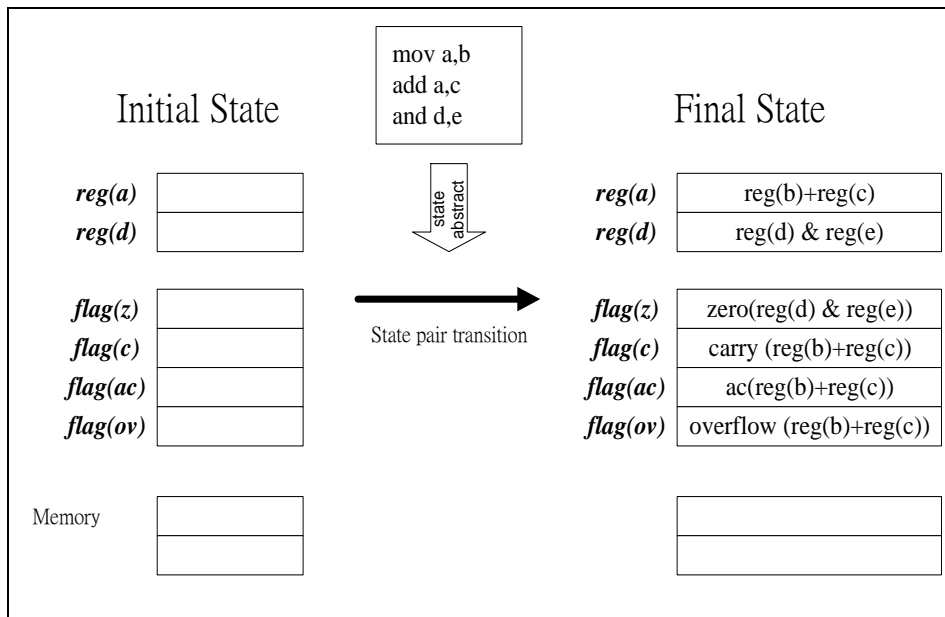


Fig. 7. State abstraction with the side effect.

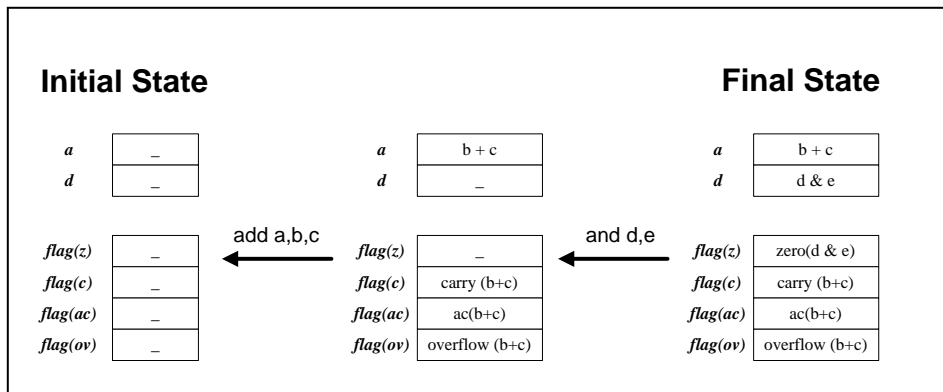


Fig. 8. Backward chaining with the side effect.

3.1.3.3 Operand data width

In this section, we consider the problem of viewing an operand as a bit.

3.1.3.3.1 Bit implementation

Dest ← SRC1 op SRC2

The previous definitions of SRC1 and SRC2 are assumed to be a architecture word, but in the 8051 architecture, SRC1 and/or SRC2 should have the ability to describe bit level implementation:

Exp: Dest[S..E] ← SRC1[S...E] op SRC2[S...E].

The description change in src and dest is content(bitsof(Loc, N1, N2), bitsof(Expression, NN1, NN2)).

In this case, the dependency check mechanism should be taken into account.

3.1.3.3.2 Modeling architecture word width

It is convenient when the source and target architecture have the same bit width, but in the real world, it is possible that the reg bit widths will not match. Therefore, in future work, we will need to model the reg width to overcome this problem. For example, assume that the source architecture is a 32-bit machine, while the target architecture is a 16-bit machine. In this case, the 32-bit machine should be performed using two 16-bit additions (assuming that an addition with carry is possible), which can be done using an instructions pair as mentioned previously.

3.2 Modeling the Retargeting Process

3.2.1 Retargeting as state transition

Fig. 9 illustrates the state transition view of assembly programs on different instruction sets. Machine I executes three instructions Op_X1, Op_X2, and Op_X3 to bring the initial state S_i to the final state S_j with S_{i1} and S_{i2} being the intermediate states. On the

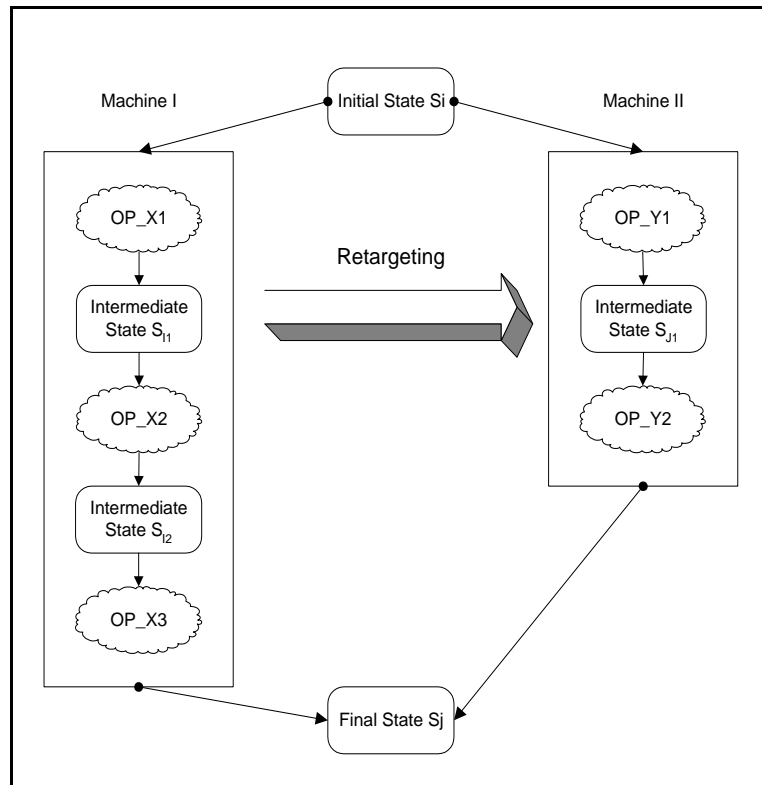


Fig. 9. Retargeting process.

other hand, machine II executes two instructions Op_Y1 and Op_Y2 to bring the machine from the same initial state S_i to the same final state S_j with S_{j1} being the intermediate state. Although with different intermediate states, the instruction sequences $\{Op_X1, Op_X2, Op_X3\}$ and $\{Op_Y1, Op_Y2\}$ bring machine I and machine II, respectively, to the same final state, as long as they start from the same initial state. Therefore, the latter sequence can be regarded as the result of retargeting the former sequence from machine I to machine II, and vice versa.

3.2.2 Applying operators and creating the intermediate states

Instructions are considered as operators which, when applied, change the state of the machine. Therefore, *the retargeting process is a process of selecting appropriate operators to bring the machine from an initial state to a final state*. Since both the operators (instructions of the target machine) and the basic blocks (of the original assembly program) are abstracted as states, the selection of operators to be applied can be regarded as a matching process. In the current implementation, we select the appropriate operator in the following manner:

1. Find the perfect match first. Both the location and value are matched.
2. Find an operator that the location matches, where the value is similar to the value part of the instruction.

3. Find an operator that the value matches and whose location is similar.
4. Find an operator whose location matches and whose value part is similar and is an expression

The operator selected by the rule 1 reduces the size of the state to be achieved (i.e., the problem state). The retargeting process is completed when the size of the problem state is zero. The operators selected by rules 2 to 4 do not reduce the size of the problem state, but create some intermediate state to which other operators can be applied. If there is no applicable operator, then the retargeting process halts.

3.2.3 Backward chaining

We adopt the backward-chaining algorithm to solve the retargeting problem. A solution can be constructed backwards in the following way: first, select an operator whose post-condition best matches the given final state; second, construct an intermediate state (a state closer to the initial state than the original final state) by deleting the post-condition from and adding the pre-condition to the original final state; third, if the intermediate state is not equal to the initial state, then it serves as the new final (goal) state, and the planned construction is repeated. An example is depicted in Fig. 10. State *Sz* is the result of applying operator *op4* backwards. In other words, applying *op4* to state *Sz* can result a state transition to a final state when we are in state *Sz*. State *Sy* is the result of applying operator *op3* backwards, and so on. After searching, we obtain the solution in the sequence *op4*, *op3*, *op2* and *op1*. The final solution is reversed, and we get *op1*, *op2*, *op3* and *op4*.

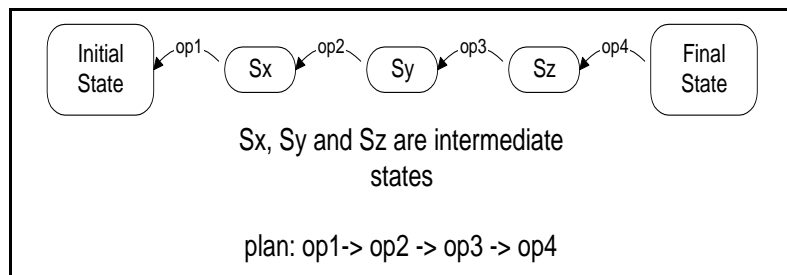


Fig. 10. Backward chaining.

Fig. 11 shows another backward chaining retargeting process illustrated as in real instruction set architecture. In Fig. 11 (a), for example, the basic block on the source architecture has the state transition as accumulator ACC will get the content of mem(a) plus the content of mem(b). Fig. 11 (b) shows that the target processor has instruction `mov ACC, mem(a)`, which says that the accumulator will get the content of mem(a) and add `ACC, mem(b)`, which says that accumulator ACC will get the content of ACC plus mem(b). In Fig. 11 (c), we see that if we apply the instructions `add` and `mov` backward, then we will have the same state transition as that shown in Fig. 11 (a). Then, we can get the retargeting result as `[mov; add]`; that is, the basic block when migrating to the target architecture will be `[mov; add]`.

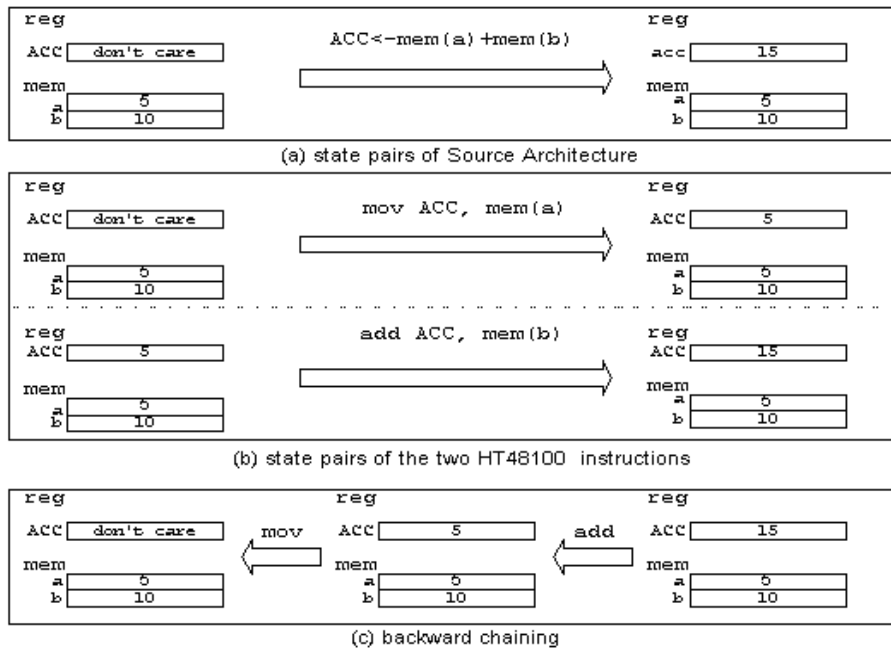


Fig. 11. Backward chaining.

3.2.3.1 Basic backward chaining algorithm for retargeting

Fig. 12 shows the backward chaining algorithm for our retargeting system. The termination condition is **Final_State == Initial_State**; before the termination condition is met, it will try every possible operator that can bring the final state closer to the initial state (line 7~8). When trying each operator, it will recursively call itself with a new final state (line 10).

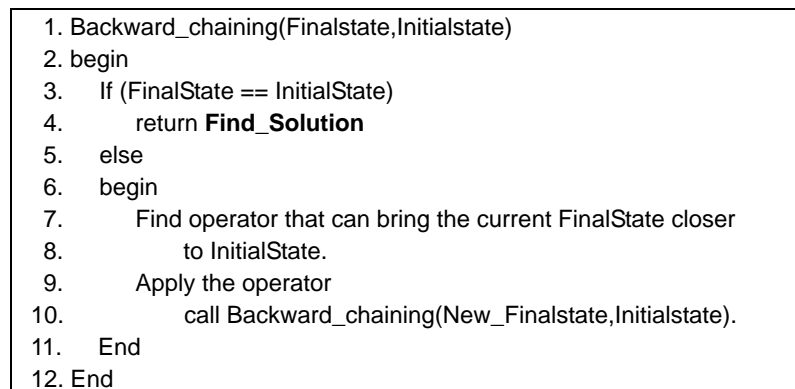


Fig. 12. Backward chaining algorithm.

3.2.4 Ordering of operators

3.2.4.1 Dependency modeling

Our state transition list representation defines the state transition of the system state. In our state list modeling, each element of the list describes the system state that will change; it doesn't indicate any sequential order relationship. The state list representation means that we have a set of state transitions. However, when we map the state list to target instruction sets, the sequence of modifications of the system state is important. Therefore, we must guarantee the correct modification sequence of the system state. In the preprocessing phase, the mechanism to augment the initial state list representation with ordering information.

3.2.4.2 Motivation

Why do we need to worry about dependence?

In the state notation, each state transition tries to bring the machine state to some desired final state. Theoretically, each state transition has no sequential relationships. In other words, the state list notation [ST1, ST2] that describes the transition of the system state does not mean that ST1 and ST2 have any sequential relationship needed to bring the machine to some desired state. It just means that they are ST1 and ST2 state transitions. However, when the state representation is mapped to target instructions, the sequence of mapped instructions is important to guarantee that modifications of the value are performed in the correct order under the target machine. Because the Retargeting Engine is a plan search algorithm, the order of the search results is important.

For example, with two state transitions [content(reg(a), reg(b)), content (reg(c), reg(a))], the first one tries to transfer the initial value of reg(b) to reg(a), and the second tries to transfer the initial value of reg(a) to reg(c). In the state notation, the two operations have no sequence problem. But if we try to generate the target code, we must define the operation sequence of the two operations because we must guarantee the sequence modifications of the system state. In the previous case, because the second state transition tries to read the initial value of reg(a) while the first state transition tries to modify it, the second one should be mapped to a real instruction before the first one is. **Therefore, the rule for mapping a state transition to an instruction sequence states that the node that modifies a location should come after the other node that reads the location as it's initial value.**

To illustrate this, for example, if the first state transition is mapped before the second one, that is, content(reg(a), reg(b)) is mapped before content(reg(c), reg(a)), then the instruction sequence will be [mov reg(a), reg(b); mov reg(c), reg(a)]. We will get the desired result as reg(a) and reg(c), and get the initial value of reg(b), which is not our expected result.

3.2.4.3 Dependence model

Since in our system state modeling, the state notation does not guarantee the sequence of the state transition, but the mapping result of the instruction sequence of the state transition needs to be in the correct instruction sequence. Therefore, we need to

construct the dependency relation of the state transition so as to guarantee the correct order of the mapping state transition to target instructions. In this section, we will construct a dependency graph for each basic block. Fig. 13 shows our modeling of a dependency graph. **The rule for mapping a state transition to an instruction sequence states that the node that modifies a location should come after the other node that reads the location as its initial value.** We will connect each node based on this dependence rule.

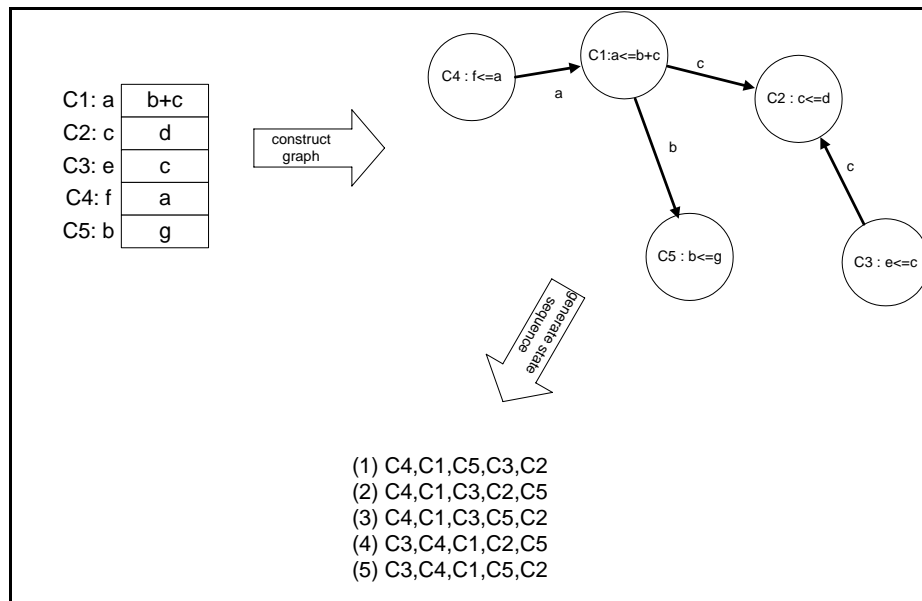


Fig. 13. Dependency graph and legal schedules.

3.2.4.4 Dependency schedule

After we have constructed the dependency relationship graph, we can easily schedule the node to decide which state transition will be mapped to the target architecture instructions first. In Fig. 13, we see that there are five possible instruction sequences that can accomplish the required state transition. For example, if we choose sequence (1) [C4, C1, C5, C3, C2], the target instruction sequence may be `mov f, a; add a, b, c; mov b, g; mov e, c; mov c, d`. If we choose sequence (5) [C3, C4, C1, C5, C2], the target instruction sequence may be `mov e, c; mov f, a; add a, b, c; mov b, g; mov c, d`. We can see that different instruction sequences may lead to the same state transition result.

3.2.4.5 Breaking up loops

Sometimes, instructions like swaps have loops when abstracted as state transition list descriptions. That is, for example, when we write the state transition of swap `a, b`, it will be `[content(a, b), content(b, a)]`. If we construct the dependency graph of the state list

it will appear as shown in Fig. 14 (b). There is a loop relation between the C1 and C2 state nodes. In this situation, the previously mentioned scheduling mechanism can not decide from the dependency graph which state transition will come before the other when mapped to a real target instruction set because the dependency between them forms a loop. Therefore, the first step of the dependency check algorithm breaks up the loop relation formed by the state transition. In the figure, we show that when we insert an intermediate state C3, we can change the dependency edge between C1 and C2, and thus, break up the loop. We can see that when we decide to break the dependency edge b, we insert the intermediate state C3. State C3 will store the initial value of b to temp storage Tb, and state C1 will be modified so as to get the final value of Tb, represented as Tb_f. In addition, the direction of the dependency edge will be from C3 to C1. We can see that there no loop exists in Fig. 14 (b).

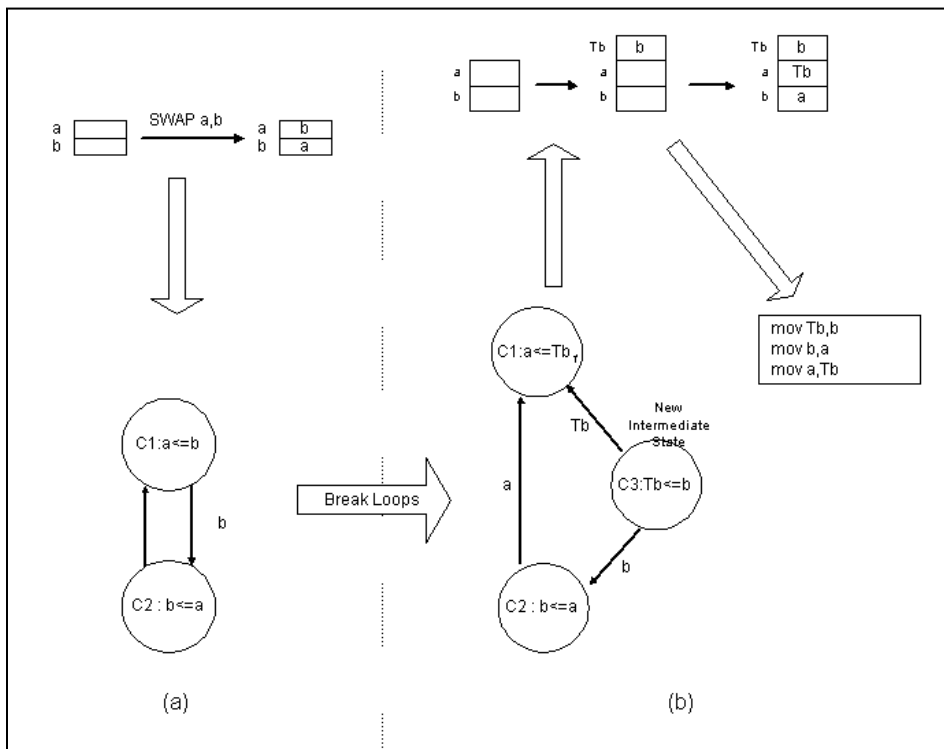


Fig. 14. Breaking up a loop in the dependency graph.

3.2.4.6 Comment on the schedule for the ready set

After the dependency relationship is constructed, we can schedule the mapping sequence of the state transition list.

The dependency resolution phase is especially suitable for the Super Scalar and Pipeline architectures because the code after dependency checking can schedule instruc-

tions so as to remove depended instructions, and issue more parallel instructions. Thus, the sequence will be executed more efficiently than the initial code in the Super Scalar and Pipeline architectures. For example, assume that there are two instruction chains that are independent of each other. If the two chains are scheduled in a parallel manner, more instructions can be issued under the Super Scalar architecture. But if they are scheduled sequentially, the instructions can not be efficiently issued under the Super Scalar architecture.

In the case of the VLIW target architecture, we can further see the importance of the dependency schedule algorithm. We know that VLIW can execute more parallel instructions concurrently. If we can schedule more parallel instructions to be executed, we can take further the advantage of the VLIW architecture.

3.3 Storage Mapping

Due to the different architectures of the source and target micro-processors, we provide a mechanism to map the difference between the architecture resources. For example, different processor architectures may organize registers file, memory, or IO architecture differently. In this case, some storage mapping must be done to ensure that the mapping process will succeed. Furthermore, there may be some dedicated storage in some processors, so different architecture properties must be described explicitly in the storage mapping step. In our tool, the storage mapping information is described in the technology file, which also describes the architectural difference between the source and target architectures. In this phase, the tool will translate the state abstraction file into a ready-to-retarget file according to the technology file. For example, when it tries to translate from the MIPS CPU to the X86 CPU architecture, because X86 has fewer general purpose registers, some registers must be mapped to memory locations. “**storagemap(reg(r2), mem(r2))**” means that reg(r2) on the source architecture will be mapped to memory location r2 on the target architecture. The following figure (Fig. 15) shows the mapping between MIPS and X86 when we wish to retarget the assembly program of MIPS to the X86 architecture. It shows that MIPS register R0 ~ R3 is mapped to x86 general register AX, BX, CX, DX. R4 ~ R31 is mapped to a memory location. The mapping information is provided by the user, who must be familiar with the source and target processor architectures.

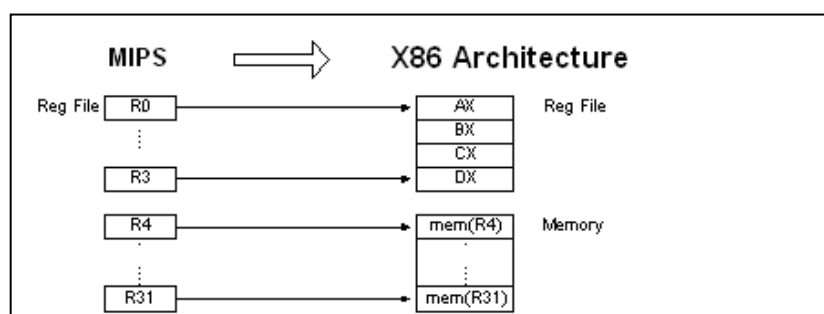


Fig. 15. Storage mapping of MIPS to X86.

Fig. 16 shows another register mapping example of retargeting from the mc8051 platform to the HT48100 platform, the required resource mapping between the two different resource architecture. Fig. 18 shows the storage mapping description file in our retargeting tool. Fig. 17 shows the state representation after storage mapping, where $reg(r1)$ and $reg(r2)$ are both mapped to memory location in the HT48100 platform.

Fig. 17 shows the state transition graph after storage mapping. Note that registers $reg(r1)$ and $reg(r2)$ are mapped to memory locations $mem(r1)$ and $mem(r2)$.

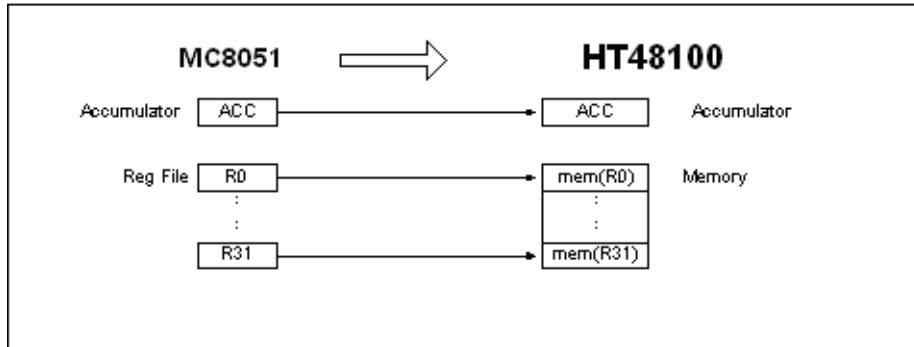


Fig. 16. Storage mapping of MC8051 to HT48100.

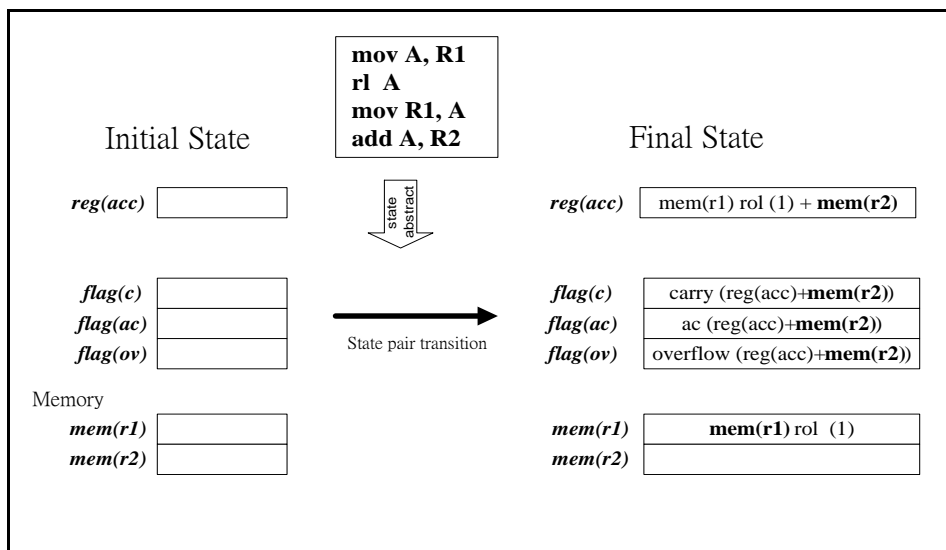


Fig. 17. State transition after storage mapping.

```

regmaptab(reg(r0),mem(immed(r0))).
regmaptab(reg(r1),mem(immed(r1))).
regmaptab(reg(r2),mem(immed(r2))).
regmaptab(reg(r3),mem(immed(r3))).
regmaptab(reg(r4),mem(immed(r4))).
regmaptab(reg(r5),mem(immed(r5))).
regmaptab(reg(r6),mem(immed(r6))).
regmaptab(reg(r7),mem(immed(r7))).
regmaptab(reg(sp),mem(immed(sp))).
regmaptab(bits(reg(psw),5),bits(mem(immed(hex0ah)),0)).
regmaptab(reg(psw),mem(immed(hex0ah))).
regmaptab(reg(dptr),mem(immed(dptr))).
regmaptab(reg(stack),mem(immed(stack))).
regmaptab(reg(pp2),mem(immed(pp2))).
regmaptab(reg(p1),mem(immed(p1))).
regmaptab(reg(p3),mem(immed(p3))).
sideeffectmaptab(s(z,_),s(z,_)).
addsideeffect([s(z,_)]).

```

Fig. 18. Storage mapping file between MC8051 and HT48100.

4. THE RETARGETING FRAMEWORK

Fig. 19 shows the flow graph of our retargeting system framework. The main retargeting phase includes state abstraction, storage/IO mapping, the retargeting engine, and human suggestions. The main input files are application programs that need to be retargeted to another instruction set platform, the instruction set specifications of the source and target architectures, storage and IO mapping files. In the figure, it is assumed that the original assembly program is given by instruction set A. The assembly program is to be translated into instruction set B, C, D, *etc.* We call instruction set A the *source* instruction set, and instruction sets B, C, D, *etc.*, the *target* instruction sets.

4.1 The Translation Flow

The retargeting granularity is based on the basic block boundary. It is assumed that the basic blocks in the assembly program have been clearly marked.

The first step, state abstraction, in translation derives the state representation for each basic block. This step requires an instruction set specification file for instruction set A. The specification is also based on the state representation.

Different processor architectures may organize registers file, memory, or IO architectures differently. Therefore, a second step, storage mapping, in the translation process is necessary to match the storage elements and their access methods between these architectures. For example, the registers in the MC8051 microcontroller have to be mapped to certain memory locations in the HT48100 microcontroller, as shown in Fig. 20.

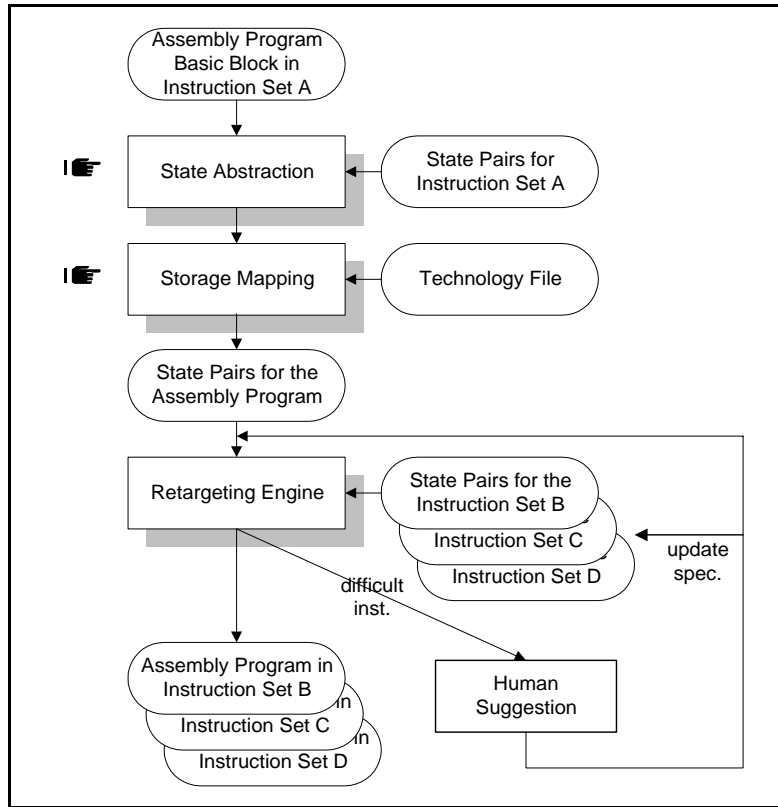


Fig. 19. The retargeting framework.

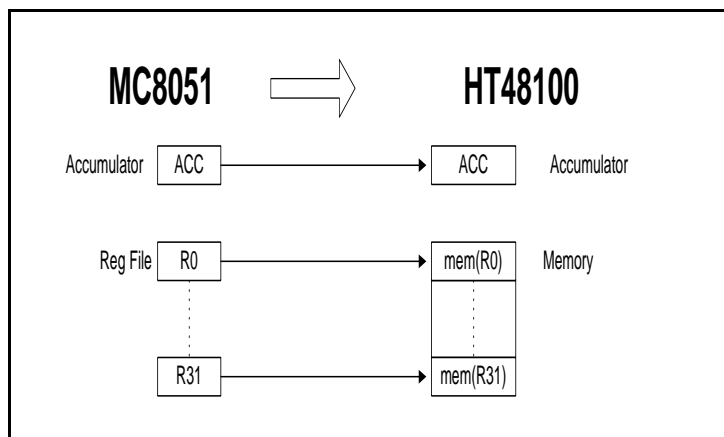


Fig. 20. Storage mapping of MC8051 to HT48100.

After the previous pre-processing steps, the basic blocks are now ready for translation. The retargeting engine, by implementing the algorithm in section 3, translates each basic block into a sequence of instructions, based on the instruction set specification of the target instruction set, which is also represented with the state notation.

4.2 The Feedback Loop in the Translation Flow

We recognize that assembly program translation is not an easy thing to automate. Therefore, we provide a manual feedback loop in the translation flow to take care of difficult cases.

One difficult case occurs when there are some instructions in the source instruction set for which the translation algorithm fails to find solutions using the target instruction set. Many of these instructions are control related. Efficient solutions for these kinds of instructions may involve some combination of more than one control flow instruction which our current algorithm fails to deal with.

For example, in MC8051, JNZ is a conditional jump under the condition that zero flag is not true. On the other hand, in HT48100, there are no conditional jump instructions. But HT48100 provides a conditional skip instruction SZ which skips the following instruction when the zero flag is true. By combining the SZ and the JUMP (unconditional jump) instructions of HT48100, we can achieve the same control flow mechanism as does JNZ of MC8051, as shown in Fig. 21. However, our current algorithm is not able to derive this combination since the basic block structure of the [SZ; JUMP] sequence is totally different from that of the JNZ instruction.

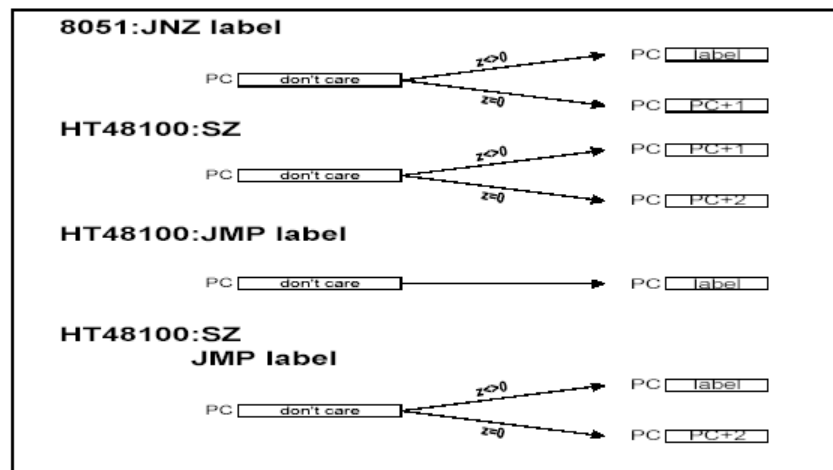


Fig. 21. Instruction pair for conditional jumps.

Once the solution (an instruction pattern) is found manually, it can be viewed as a powerful instruction and added to the instruction set specification for the target instruction set. The next time the state of the source instruction is encountered, the algorithm automatically select the known solution.

Understanding that assembly translation is by no means an easy problem, we do not expect that the entire program can be automatically translated. Instead, we aim to take care of as much as possible of the program that can be automatically translated, usually more than 85% of the entire program in our experience, so that the designer can focus on manually translating the most challenging portion through the feedback pass of the translation flow. Moreover, the manually derived solutions can be added to the target instruction set specification, so that the next time the same situation is encountered, the algorithm will know how to deal with it. As more solutions are accumulated into the specification, less manual intervention will be required.

4.3 Verification of Instruction Mapping

We can verify our mapping results through the state transition. If the produced plan can lead to the same state transition from the initial state to the final state, it is the correct mapping. In other words, the machine will have the same behavior when applying the original application and the retargeting result. Fig. 22 shown the verification process. For example, when we retarget the Intel 8051 application to the target architecture Holtek HT48100 microcontroller, we can use the plan we produced to perform the simulation to see if it is correct. We apply each operator sequentially to the target architecture and check the transition from the initial state to the final state. If the two set of states have the consistency, the codes we generated are verified and are viewed as the correct retargeting.

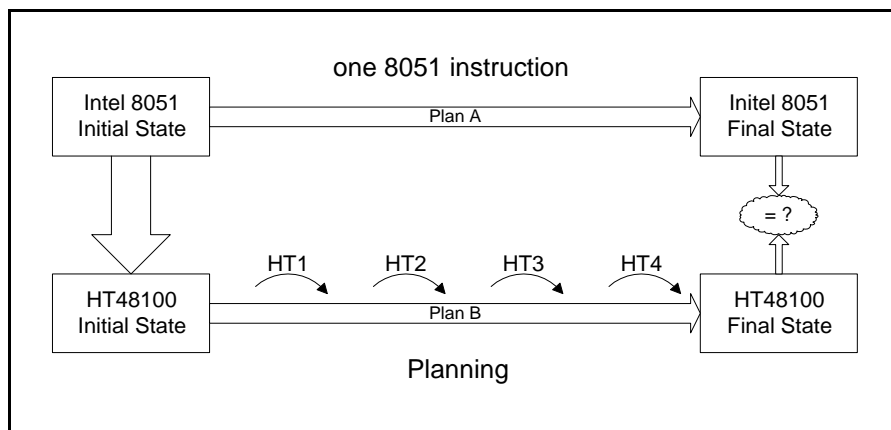


Fig. 22. Verification framework.

Another verification method uses our retargeting tools. We can use our state abstract phase as the verifier. For example, we can abstract the retarget result and compare it with the state abstraction of the source program. If their state abstractions match, then we can say that the result is verified.

Figs. 23 and 24 show the verification method that uses our retargeting tools.

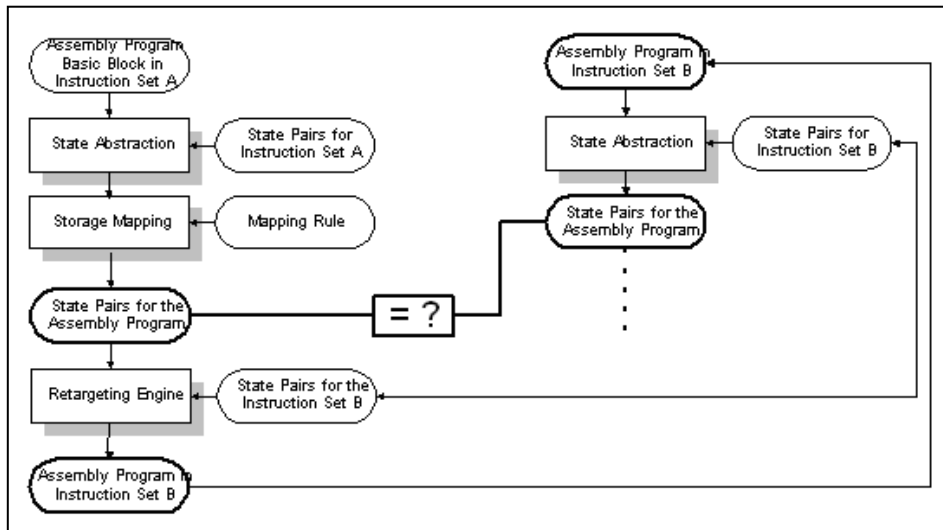


Fig. 23. Verification using retargeting tools (I).

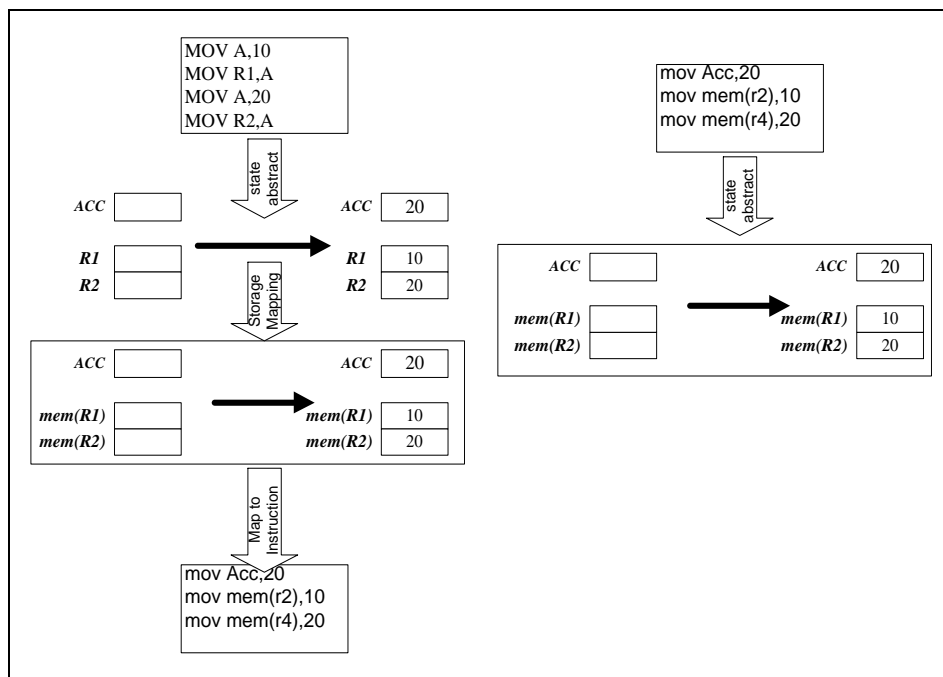


Fig. 24. Verification using retargeting tools (II).

5. EXPERIMENTAL RESULTS

The proposed instruction set retargeting technique was tested in three experiments. In section 5.1, we show the inherent optimization benefit of state abstraction of basic blocks (discussed in section 3.1.2). In section 5.2, we describe our experiment on mapping an MC8051 (with a CISC-like instruction set) assembly program to HT48100 (with a RISC-like instruction set). Finally, in section 5.3, we show that the retargeting system can be easily adapted to perform mapping between different instruction sets, such as PIC (RISC-like), MC8051, and HT48100. All the microcontrollers used in the experiments had 8-bit data paths.

5.1 Example Mapping: HT48100 to HT48100 With Optimization

To illustrate the inherent optimization benefit of our state abstraction, we abstracted three basic blocks in the HT48100 instruction set into machine states and then mapped the states back to HT48100 for comparison, as shown in Table 1. The first column lists the case numbers. The second column presents the original codes. The third column shows the translated optimized code.

Table 1. Inherent optimization due to state abstraction.

Case	Original code	Optimized code
1	Mov ACC, mem(m1) Mov mem(m1), ACC	Mov ACC, mem(m1)
2	Mov ACC, mem(m1) Mov ACC, mem(m2)	Mov ACC, mem(m2)
3	Mov ACC , immed(10) Mov mem(m1), ACC	Mov ACC, immed(10) Mov mem(m1), immed(10)

The first case shows that the redundancy created by moving data around is automatically removed during the retargeting process. The second case shows that the ACC is written twice, and that only the effect of the second instruction remains, so the first instruction's effect is automatically eliminated. The third case shows that the true dependency relationship on ACC is removed, that the code is removed, and that the code is improved after the retargeting process.

5.2 Example: MC8051 (CISC-like) to HT48100 (RISC-like)

Table 2 shows an example of mapping an MC8051 keyboard-scan program to HT48100. The second column shows the MC8051 program, partitioned into basic blocks. The third column lists the mapping result of each basic block on the HT48100 platform. The basic blocks 4, 6, 8, 11, 13, 14 and 16 contained MC8051 instructions with richer semantics and were, thus, mapped into a larger number of HT48100 instructions.

There was a complex MC8051 instruction CJNE in basic block 16. This instruction performed multi-way branching, depending on the result of comparison (greater, less, or equal). Multiple conditional states were necessary to model this instruction. Each state was mapped to several HT48100 instructions. We can see that two labels, `exit1` and `exit2`,

Table 2. Mapping from MC8051 assembly code to HT48100 code.

Basic Block ID	Processor : MC8051 (Original Code)	Processor:HT48100 (Translated Code)
1	START: MOV SP, #30H	mov mem_sp, 030h
2	MAIN: CLR PSW.5	Clr mem_psw.5
3	S1: ACALL KSCAN	Call kscan
4	JNB PSW.5, S1	Snz mem_psw.5 Jump s3
5	S2: ACALL KSCAN	Call kscan
6	JB PSW.5, S2	Sz mem_psw.5 Jump s2
7	MOV P1,A LJMP MAIN	mov mem_p1, a jmp main
8	KSCAN: JB PSW.5,S3	Sz mem_psw.5 Jump s3
9	MOV R1, #0FEH MOV R4, #04H MOV A, R1 MOV R2, #00H	Mov mem_r1, 0feh Mov mem_r4, 04h Mov a, mem_r1 Mov mem_r2, 00h
10	COLLUM: MOV PP2, A MOV R3, #04H MOV A, PP2 ANL A, #0F0H MOV 20H, A	mov mem_pp2, a mov mem_r3, 04h mov a, mem_pp2 and a, 0f0h mov mem_20h, a
11	JUG: JB ACC.4, NT1	Sz [05H].4 Jump nt1
12	MOV A, R2 SETB PSW.5 RET	Mov a, mem_r2 Set mem_psw.5 Ret
13	NT1: INC R2 RR A DJNZ R3, JUG	Inc mem_r2 rr [05H] dec mem_r3 sz mem_r3 jump jug
14	MOV A, R1 RL A MOV R1, A DJNZ R4, COLLUM	Mov a, mem_r1 rl [05H] mov mem_r1, a dec mem_r4 sz mem_r4 jmp collum
15	RET	ret
16	S3: MOV A, #00H MOV P1, A MOV A, R1 MOV PP2, A MOV A, PP2 ANL A, #0F0H CJNE A, 20H,S4	mov a, 00h mov mem_p1, a mov a, mem_r1 mov mem_pp2, a mov a, mem_pp2 and a, 0f0h mov mem_tempA, a sub a, mem_20h mov a, mem_tempA sz reg(c) jmp exit1 sz reg(z) jmp exit1 jmp s4 exit1: mov mem_tempB, a sub a, mem_20h mov a, mem_tempB snz reg(c) jmp exit2 sz reg(z) jmp exit2 jmp s4 exit2:
17	JMP S3	Jump s3
18	S4: CLR PSW.5 MOV A, R2 RET	Clr mem_psw.5 Mov a, mem_r2 Ret

were automatically inserted into the target assembly program by our retargeting engine. This case demonstrates one of the challenges in assembly translation: the basic block structure might not be preserved during translation. Basic block 16 of the MC8051 assembly, containing seven instructions, was mapped into twenty-two HT48100 instructions, which corresponded to ten basic blocks.

5.3 Mapping Between Multiple Instruction Sets

We also conducted an experiment in which several assembly programs were mapped from PIC and MC8051 to HT48100, as shown in Table 3. Since PIC and HT48100 have similar RISC architectures, the translated code was the same size as the original code. On the other hand, the code size expanded when a MC8051 (CISC-like) code was translated into HT48100 code.

Table 3. Mapping between PIC, MC8051 and HT48100.

Programs	Source Architecture	Source Assembly Lines	State Pairs	Target Architecture	Target Assembly Lines
1.LED display	PIC	25	27	HT48100	25
2.Random Number	PIC	33	35	HT48100	33
3.LED Light	PIC	37	43	HT48100	37
4.Keyboard-scan	MC8051	42	48	HT48100	65
5.LED display	MC8051	13	25	HT48100	25
6.TrafficControl Signal Light	MC8051	45	72	HT48100	63
7.Timer	MC8051	21	38	HT48100	45

Since PIC and HT48100 have similar architectures, there was no need for human help; i.e., no feedback loop in the translation flow shown in Fig. 19 was needed. On the other hand, some human intervention was necessary to aid translation from MC8051 to HT48100. After eight instruction patterns (as discussed in section 4.2) were added into the target instruction set specification, all the assembly programs could be completely (100%) and automatically translated, as shown in Fig. 25.

6. CONCLUSIONS

We have proposed a new approach to translating assembly programs between different microcontrollers. In our approach, the specifications of both the source and target instruction sets are represented as machine state transitions. The assembly program to be translated is first divided into basic blocks. Each basic block is represented as a machine state transition. The retargeting process is then modeled as selecting appropriate operators (instructions in the target instruction set) to bring the target microcontroller from the same initial state to the same final state as the original program in the source microcontroller. The state notation serves as a useful canonical representation of both the instruction sets and the assembly programs. In addition, many code optimizations are implicitly achieved during the state abstraction process.

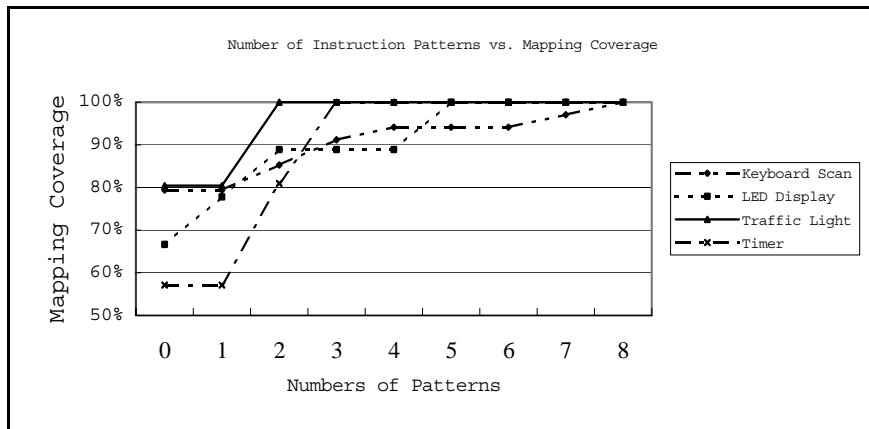


Fig. 25. Number of instruction patterns vs. mapping coverage (MC8051 to HT48100).

The proposed technique has been successfully demonstrated by translating various assembly programs between three industrial 8-bit microcontrollers, including both RISC and CISC styles.

In the future, we will investigate automatic verification of translation results based on the same state notation. In addition, transformation of the basic block structures is necessary to check if two groups of basic blocks produce the same machine state transition.

REFERENCES

1. W. F. Kao and I. J. Huang, "Instruction retargeting based on the state pair notation," *Asia Pacific Conference on Hardware Description Languages*, 1997, pp. 114-120.
2. A. V. Aho, M. Ganapathi, and S. W. K. Tjiang, "Code generation using tree matching and dynamic programming," *ACM Transactions on Programming Languages and Systems*, Vol. 11, 1989, pp. 491-516.
3. A. Chernoff et. al., "FX!32 a profile-directed binary translator," *IEEE Micro*, 1998, pp. 56-64.
4. P. Marwedel, "Tree-based mapping of algorithm to predefined structures," *International Conference on Computer-Aided Design*, 1993, pp. 586-593.
5. R. L. Sites et. al., "Binary translation," *Communication of the ACM*, 1993, pp. 69-81.
6. M. Corazao, M. Khalaf, L. Guerra, M. Potkonjak, and J. Rabaey, "Instruction set mapping for performance optimization," in *Proceedings of International Conference on Computer Aided Design*, 1993, pp. 518-521.
7. C. Liem, P. Paulin, M. Cornero, and A. Jerraya, "Industrial experience using rule-driven retargetable code generation for multimedia applications," *TIMA Laboratory and Central R&D*.
8. P. Marwedel and G. Goossens, *Code Generation for Embedded Processors*, Kluwer Academic Publisher, 1995.

9. C. Cifuentes, "Partial automation of integrated reverse engineering environment of binary code," in *Proceedings of Third Working Conference on Reverse Engineering, IEEE-CS Press*, 1996, pp. 50-56.
10. C. Cifuentes and S. Sendall, "Specifying the semantics of machine instructions," in *Proceedings of the International Workshop on Program Comprehension*, 1998, pp. 126-133.
11. N. Ramsey and M. Fernández, "Specifying representations of machine instructions," *ACM Transactions on Programming Languages and Systems*, Vol. 19, 1997, pp. 492-524.
12. C. Monahan and F. Brewer: "Symbolic modeling and evaluation of data paths," in *Proceedings of ACM/IEEE 32nd DAC*, 1995.



Ing-Jer Huang (黃英哲) received the BS degree in electrical engineering from National Taiwan University, Taiwan, R.O.C., in 1986, and the MS and Ph.D. degrees in computer engineering from the University of Southern California, U. S. A., in 1989 and 1994, respectively.

He is currently a professor in the Department of Computer Science and Engineering at National Sun Yat-Sen University, Taiwan, R.O.C. His research interests include microprocessors, design automation, system software, embedded systems, and hardware/software co-design. Many of his techniques have been successfully applied to the design of industrial microprocessors, such as Teledyne's TDY-43, Holtek's HT48x00, x86 compatible processors, ATCHIP's 8-bit multimedia enhanced microcontroller, and ACARD's and RDC's 32-bit embedded microprocessors. He also serves as a consultant to IC companies. He is a member of IEEE and ACM.



Dao-Zhen Chen (陳道正) received the B.S. degree in Electrical Engineering from FengChia University, Taiwan, R.O.C., in 1996 and the M.S. degree in computer engineering from National Sun Yat-Sen University, Taiwan, R.O.C., in 1998. His research interests include system software designs. He is currently an SW design engineer at the SW design department of MicroLinks Technologies, Inc.