Chapter 9

Optimisation

In this chapter, we will study many methods to improve the performance of the system that we have built in the previous chapters. The system consists of some simple applications running on a concurrent operating system written in a high level language. The platform (the hardware system, a virtual one) is a stack-based processor. It is microprogrammable, so its instruction set can be extended. The high level language itself can also be changed. We have written the compiler for the language and the code generator to generate code for the target machine. These programs can be modified.

Studying a computer system as a whole can reveal the relationship between components. Their interactions can be complex and interesting. The optimisation aims to improve mainly the performance, to complete a task with fewer numbers of cycles. As we built all components by ourselves, we can change the system at every level. We can experiment with any component and observe the change. We can instrument our system to collect statistics easily.

There is no separate *lab session* in this chapter as the work is spread out in all sections. Each optimisation method will be tried and data collected. The analysis follows each experiment.

9.1 Framework

There are many levels in which to aim an optimisation for. The highest level is an algorithmic level. We will not explore this topic; instead we refer to many excellent textbooks in the field [COR01] [PRA01] [KLI05]. What we will explore is at a more concrete level: the language level, the code generation level and the microprogramming level.

At the language level, the macro expansion will be studied. The extension of the language itself to include new operators will be tried. At the code generation level, many techniques will be investigated: supporting new instructions such as increment, decrement (which are implemented in the previous chapter), improving some simple sequence of code, and eliminating some code. At the instruction level, a few new instructions will be designed and implemented. The microarchitecture level has been demonstrated in the chapter 7, showing an improving data path of the Sx processor.

What are we going to measure and how?

To observe any improvement we need to set up a controlled environment. Several methods will be applied to improve a system that performs the same task. The effects of these different methods can then be observed and compared. The benchmark programs are a set of programs (or tasks) representing the kind of workload that we expect in our work. We elected two programs as our workload representatives: the Nut compiler and the Nut operating system.

The first benchmark is the compiler benchmark. The original Nut compiler, “nut.txt”, is used to compile itself. This represents a substantial work that is moderately complex and contains many well-known problems in computer science. Nut compiler is also a non-trivial program that will exercise a large repertoire of instructions.

The second benchmark is the message passing benchmark. An application program performs producer/consumer type of behaviour. It sends and receives 100 simple messages. This benchmark tests the operating system, the task switcher and represents the fundamental operation in the operating system, the interprocess communication. The program is “nos2.txt”. It contains Nos and the messaging services: send, receive. There are 200 task switches.

The data collection consists of the profile of running the benchmark programs. Two statistics are collected:

1. Frequency of each instruction used
2. Frequency of each line of program used

The statistic 1 let us know what instruction to improve. The statistic 2 let us know where the programs spend its time.

Tools

The set of tools that will be used are:

1 The original Nut compiler, the source is “nut.txt”; the N-object code is “nut.obj”.
2 The code generator, the source is “gen.txt”; the N-object is “gen.obj”.
3 The evaluator of N-object, the Nut virtual machine, nvm. It is an executable code running on a real computer.
4 The Sx processor, its simulator is used to execute the S-object which is considered to be the “grounded” level for measuring the number of cycle used to run benchmark programs.

Baseline

We establish the *baseline* data to be compared with the result of the methods suggested in this chapter.

This is the profile of the compiler benchmark. Let “nuts.obj” be the S-code of Nut compiler, “nut.txt”. We run the following task to collect the statistic.

```
c:>sx nuts.obj < nut.txt
```

The base Nut compiler, compiles the original Nut compiler. “`sx`” will output a profile file, “prof.txt” showing the frequency of each instruction used and the frequency of each line of program used. Table 9.1 shows the profile of the number of instruction used the functions. Only the functions that consume more than 100 (x1000) instructions are shown. The total number of instruction executed in this benchmark is 8585 (x1000).

Table 9.1 The number of instruction (x1000) used in functions in the compiler benchmark (anything less than 50 is not shown)

!=	738
and	963
str=	4414
getName	416
install	1456
	7987

Observation

In terms of functions that consume most of cycle, the “str=” is the first one, followed by “install”, “!=“, “and”, “getName”. They are summed up to 93% of total instruction executed. The “str=”, string comparison function, alone consumes 50% of cycle. This function is used almost entirely on the task related to the symbol table. This fact suggests that we should concentrate on supporting this function in machine instructions. We should also consider changing the access methods of the symbol table.

The next step is to collect the data of the message passing benchmark. The profile of running nos2 is shown in Table 9.2.

They are accounted for 82% of total number of instructions executed. The total number of instruction executed in this benchmark is 43580.

The functions in the table are mostly the functions accessing data structure in the form (vec a n), (setv a n), where a is a local variable, n is a constant.

Table 9.2 The number of instruction used in functions (anything less than 1000 is not shown) in the message passing benchmark.

!=	1993
or	1295
ei/di	1194
getNext	1992
setNext	2020
setPrev	2020
appendDL	3413
deleteDL	2700
setValue	3005
switchp	3804
findmail	3465
send	3267
receive	2772
produce	1581
consume	1395
	35916

9.2 Macro expansion

To reduce the overhead of a function call, a function can be defined as a "macro". A macro definition is just like a function definition, the difference is that the body of a macro definition is substituted into the call. Hence, the size of a program with a macro is larger. The advantage is that it will be executed faster. The syntax of a macro definition is similar to a function definition, only the keyword is "defm" instead of "def". For example,

```
(defm print (x) () (sys 1 x))
```

Whenever the macro appears in the program, the macro body is substituted.

```
(def report (a) ()
  ...
  (print a)
  ...
```

will become

```
(def report (a) ()
  ...
  (sys 1 a)
  ...
```

Macro is suitable for defining the access function such as the following functions (from symbol table access functions in “nut.txt”)

```
(def getName idx () (vec symtab idx))
(def getType idx () (vec symtab (+ idx 1)))
(def setName (idx nm) () (setv symtab idx nm))
(def setType (idx ty) () (setv symtab (+ idx 1) ty))
```

These functions will be executed much faster because there is no overhead associated with “call” and “return” such as create/destroy the stack frame. Our macro definition cannot have local variables because the local variables in the body of the macro must be appended to the list of local variables of the caller, that is, extending the environment of the caller. We opt to demonstrate only a simple macro substitution without changing the caller environment.

The Nut-language is extended to have macros. The new keyword “defm” is recognised by the extended compiler, “nut4.txt”. The macro definition will be parsed as a normal function definition, only that its type will be “macro” (instead of “func”). The expression that called the macro can be recognised by inspecting its type. For the macro call, the body of macro will be expanded with the proper binding of actual parameters to the formal parameters defined in the macro definition.

The main function of the macro expansion is the function “subst”. The function “subst” takes apart the body of macro definition one by one element, and maps that item to the corresponding element in the actual parameter list using “mapATOM”.

```
; do macro expansion
(def domacro (nm e1) (arg body)
   (do
   (set arg (de_arg nm))
   (set body (pick (getVal arg) 2))
   (subst body e1)))

; e1 is the body of macro def, e2 is the actual arg list
(def subst (e1 e2) (e)
   (if (= e1 NIL) NIL
      ; else
      (do
      (set e (head e1))
      (if (isATOM e)
         (cons (mapATOM e e2) (subst (tail e1) e2))
         ; else
         (cons (subst e e2) (subst (tail e1) e2))))))
```

Where (pick e n) gets the n-th element of the list e. Most of the work is done in "mapATOM", where the term "get.a" or "put.a" in the body of macro definition is substituted with the corresponding actual parameters from the argument list. The following rules are the rules for substitution:

```
mapATOM a e2
   e3 = (pick e2 n)
   if a = get.n  out e3
   if a = put.n
     if e3 = get.n out put.n   local
     if e3 = ld.n  out st.n    global
   if a = ldx.n
     if e3 = get.n out ldx.n   local
     if e3 = ld.n  out ldy.n   global
   if a = stx.n
     if e3 = get.n out stx.n   local
     if e3 = ld.n  out sty.n   global
   otherwise
     out a                     do not substitute
```

Where a is the atom in the macro body, e2 is the argument list of the caller. The function (pick e n) select n-th element of e.

The function "mapATOM" and its related functions are shown below.

```
; return n-th element of e
(def pick (e n) ()
   (if (= e NIL) NIL
   (if (< n 1) NIL
   (if (= n 1) (head e)
   (pick (tail e) (- n 1))))))

; return op1/op2 depends on e3 is get/ld
(def map2 (e3 op1 op2) (e n)
   (do
   (set n (de_arg e3))
   (if (isOp e3 xGET)(set e (mkATOM op1 n))
   (if (isOp e3 xLD)(set e (mkATOM op2 n))
   (error "no ld/get in caller macro expansion")))
   e))

; map atom a with n-arg in e2, e2 is the actual arg list
(def mapATOM (a e2) (e e3 c n)
   (do
   (set c (de_op a))
   (set n (de_arg a))
   (set e3 (pick e2 n))                ; n-arg of caller
   (if (= c xGET) (set e e3)
   (if (= c xPUT) (set e (map2 e3 xPUT xST))
   (if (= c xLDX) (set e (map2 e3 xLDX xLDY))
   (if (= c xSTX) (set e (map2 e3 xSTX xSTY))
   (set e a)))))                       ; no substitution
   e))
```

Example

```
(defm inc2 (a b) ()
   (a = a + b))
```

The expression (inc2 x y) will be expanded as follows. The macro body is:

(put.a (add get.a get.b))

The atoms in the macro body are: 1) put.a 2) add 3) get.a 4) get.b. Let the arguments be two cases: locals, globals, then (inc2 x y) is compiled to:

1. x, y are locals, the call expression is (call.inc2 get.x get.y)

   The substitution for each atom is as follows.
   atom put.a map to get.x, out put.x
   atom add don't map, out add
   atom get.a map to get.x, out get.x
   atom get.b map to get.y, out get.y

   The output is (put.x (add get.x get.y))

2. x, y are globals, the call expression is (call.inc2 ld.x ld.y)

   The substitution for each atom is as follows.
   atom put.a map to ld.x, out st.x
   atom add don't map, out add
   atom get.a map to ld.x, out ld.x
   atom get.b map to ld.y, out ld.y

   The output is (st.x (add ld.x ld.y))

Similarly for the atom "ldx" and "stx". The atom in the actual parameters can not be global as it is a call by value.

How to do macro

The Nut-compiler ("nut.txt") is modified to expand macro ("nut4.txt"). The compiler itself (the target) has changed almost all the access functions to be the macros ("nutm.txt"). The compiler, nut4.txt, is used to compile the macro-version of compiler ("nutm.txt"). Then the macro-version compiler is used to compile the original, "nut.txt" for benchmarking. This is the step of the work.

1 Produce the compiler that can expand macro.

```
c:>nvm nut.obj < nut4.txt > nut4.obj
```

2 Use this compiler to compile the macro-version of compiler

```
c:>nvm nut4.obj < nutm.txt > nutm.obj
```

3 Generate S-code for “nutm.obj”.

```
c:>nvm gen.obj < nutm.obj > nutms.obj
```

The final executable code is the compiler with most access functions inlined, “nutms.obj”. We use this compiler to compile the original Nut-compiler to collect profiling statistics.

4 Run the compiler benchmark

```
c:>sx nutms.obj < nut.txt
```

The second benchmark is similar.

1 First, compile the macro-version of Nos, “nos2m.txt”.

```
c:>nvm nut4.obj < nos2m.txt > nos2m.obj
```

2 Produce the executable code.

```
c:>nvm gen.obj < nos2m.obj > ns2.obj
```

3 Run Nos (with most access functions inlined) under Noss.

```
c:>noss ns2.obj
```

The result from macro expansion (inline) to eliminate calls is shown in Table 9.4. In the compiler benchmark, using macro is 30% faster (in terms of cycle), and 46% faster in the message passing benchmark.

Table 9.3 The frequency of each instruction used

	compiler instr. X1000					message passing instr.				
	base	macro	prim	codegen	extend	base	macro	prim	codegen	extend
Add	244	244	244	121	121	107	107	107	9	9
Sub	4	4	4	4	4	1	1	1	1	1
Mul	1	1	1	1	1	0	0	0	0	0
Div	0	0	0	0	0	0	0	0	0	0
Band	13	13	227	227	227	0	0	0	0	0
Bor	0	0	2	2	2	0	0	200	200	200
Bxor	0	0	0	0	0	0	0	0	0	0
Not	0	0	0	0	0	0	0	0	0	0
Eq	379	275	256	228	220	1496	1397	1197	798	400
Ne	0	0	123	120	0	0	0	299	0	0
Lt	112	111	112	112	112	198	198	198	198	198
Le	0	0	0	0	0	0	0	0	0	0
Ge	0	0	0	0	0	0	0	0	0	0
Gt	1	1	1	1	1	0	0	0	0	0
Shl	2	2	2	2	2	0	0	0	0	0
Shr	4	4	4	4	4	0	0	0	0	0
Mod	0	0	0	0	0	0	0	0	0	0
Ldx	364	364	364	364	107	1094	1094	1094	1094	0
Stx	6	6	6	6	1	1727	1727	1727	1727	0
Ret	629	132	289	289	289	5896	1105	5397	5397	5397
Array	1	1	1	1	1	3	3	3	3	3
End	0	0	0	0	0	203	203	203	203	203
Get	2774	1949	2201	2078	863	12613	6608	11714	11616	8795
Put	824	824	824	701	137	1694	1694	1694	1596	1596
Ld	121	121	121	121	121	1812	1713	1812	1812	1812
St	3	3	3	3	3	414	414	414	414	414
Jmp	355	355	229	220	115	1098	1098	800	401	401
Jt	339	339	339	352	231	496	496	496	496	895
Jf	620	620	281	268	124	1597	1597	1098	1098	301
Lit	1126	1021	897	742	398	5947	5848	5549	4753	1932
Call	629	132	289	289	289	5896	1105	5397	5397	5397
Inc	0	0	0	123	3	0	0	0	98	98
Dec	0	0	0	0	0	0	0	0	0	0
Sys	23	23	23	23	23	1288	1288	1288	1288	1288
Jne	0	0	0	0	7	0	0	0	0	398
Ldxv	0	0	0	0	230	0	0	0	0	1094
Stxv	0	0	0	0	5	0	0	0	0	1727
Seqi	0	0	0	0	120	0	0	0	0	0
total	8585	6556	6855	6412	3777	43580	27696	40688	38599	32559

Table 9.4 The profile of the benchmarks, comparing the baseline and the macro expansion. (1) is macro/base

	compiler		message passing		
	instr. x1000	cycle x1000	instr	cycle (system:user)	cycle/ switch
base	8585	38033	43580	220915 (49274:171641)	246
macro	6556	26422	27696	120644 (22724:97920)	114
(1) %	76.4	69.6	63.6	54.6	

Observing the frequency of instruction used in Table 9.3, the "call" (plus "ret") is reduced by 80% in macro version and 30% of "get" is reduced (in getting the parameters). Similar reduction is observed in Nos benchmark, 80% reduction in "call", and 48% reduction in "get". So macro expansion is highly effective.

9.3 Introduce new primitives into the language

Nut is designed to be minimal. Many basic and frequently used functions are written as user-defined functions, such as !=, >=, <=, and, or, not. As the processor Sx already has machine instructions to support these functions, they should be considered as built-in operators of the language. The code generator can be modified to convert the call to these functions into generating the associated instructions of the Sx processor. This is not the same as macro expansion because the way Sx instruction behave is not exactly the same as the same operation written in Nut-language, for example the "and" function, in Nut.

```
(def and (a b) (if a b 0))
```

The meaning of this "and" is "if *a* is true then the result depends on *b*, else the result is false". Its semantic is the "short-cut and" where the argument is evaluated enough to know the result (not always evaluate all arguments as in *eager evaluation* semantic). However, the machine instruction "Band" will evaluate all arguments. So there is a difference. The macro expansion will

preserve the semantic of the function. The machine primitive will be faster but not always. In some case where evaluating arguments is costly, “short cut” semantic may be faster because it may evaluate less number of arguments.

Here is how the code generator is modified. The functions !=, >=, <=, and, or, not are still written as user-defined functions in the source program, their use will be compiled into function calls. The code generation for “call” checks the index to these functions and generates Sx machine instructions instead of a normal call.

```
; e is arglist
(def gencall (arg e) (idx a)
   (do
   (set idx (searchRef arg))
   ...
   (if (= idx yNe) (genop icNe 0 e)
   (if (= idx yLe) (genop icLe 0 e)
   (if (= idx yGe) (genop icGe 0 e)
   (if (= idx yAnd) (genop icBand 0 e)
   (if (= idx yOr) (genop icBor 0 e)
   (if (= idx yNot) (genop icNot 0 e)
   ; else
   (genop icCall idx e)))))))))        ; normal call

; eval arg-list (e), out code op.arg
(def genop (op arg e) ()
   (do
   (while e
      (do
      (eval (head e))
      (set e (tail e))))
   (outa op arg)))
```

Where yNe, yLe, yGe, yAnd, yOr, yNot are the indexes to the user-defined functions !=, <=, >=, and, or, not. These indexes are found in the symbol table which is read by the code generator. The instructions icNe, icLe, icGe, icBand, icBor, icNot are the native Sx instructions for not-equal, less-than-or-equal, greater-than-or-equal, bitwise-and, bitwise-or, logical-not operations.

For example the following code fragment from “nut.txt” (the last line of function “str=”)

```
...
(and (= c1 0) (= c2 0))))
```

is normally compiled to call to the defined function “and”.

```
133 Get 1
134 Lit 0
135 Eq
136 Call and
```

With this code generator it is compiled into:

```
133 Get 1
134 Lit 0
135 Eq
136 Band
```

Let this code generator be “gen5.txt”. The following steps produce the profile statistic.

1 Compile the code generator.

```
c:>nvm nut.obj < gen5.txt > gen5.obj
```

2 Generate the compiler.

```
c:>nvm gen5.obj < nut.obj > nutp.obj
```

3 Run “nutp.obj” to compile “nut.txt”.

```
c:>sx nutp.obj < nut.txt
```

Similarly for the operating system, the step of work is as follows.

1 Generate the operating system using the modified code generator, “gen5.obj”.

```
c:>nvm gen5.obj < nos2.obj > nos2p.obj
```

Table 9.5 The profile of the benchmarks, comparing the baseline and the primitive. (1) is primitive/base

	compiler		message passing		
	instr. x1000	cycle x1000	instr	cycle (system:user)	cycle/ switch
base	8585	38033	43580	220915 (49274:171641)	246
primitive	6855	28987	40688	206450 (44468:161982)	222
(1) %	79.8	76.2	93.3	93.4	

2 Run “nos2p.obj” to collect statistics.

```
c:>noss nos2p.obj
```

The result of running benchmarks using the code generator “gen5.txt” is shown in Table 9.5.

In terms of speedup (cycle), using only primitives is not as effective as macro expansion as it is only 24% faster (macro is 30%) but they are comparable. However, in the message passing benchmark, the primitives are not used much, the speedup is only 7% and 8% in task switching (for macro, 46% and 50%). So using primitives is not effective in the operating system as much as using macro expansion.

Inspecting the compiler task profile revealed that the reduction in the number of “call” is 56%, and “get” is 21%, not as much as in macro expansion (89% and 48% consecutively). In the message passing benchmark, the native instructions “Bor” and “Ne” generated by the modified code generator, are executed only 499 times, merely 1.2% of the total number of instruction executed.

9.4 Improving the quality of code from the code generator

The next step, the quality of code from the code generator can be improved. This method can be applied without changing the instruction set or the language. Some sequence of operations can be replaced by a shorter sequence of operations without affecting the result, hence making them faster.

We elected to do the following code optimisation, which are not difficult to implement.

1 Introduce inc/dec local variables as there are native instructions supported in Sx processor (as done in the exercise 6.4 of Chapter 6).
2 Improve jmp to jmp, jmp to ret, to "short cut" them.
3 Change (!= p 0) to p.
4 Change the conditional (= a 0) in "if" expression, there are two possibilities.
  4.1 (if (= a 0) x y) is replaced by (if a y x)
  4.2 (if (= a 0) x) is replaced by (if a skip x)

The expression with inc/dec can be detected from the N-code.

```
(set a (+ a 1))
```

It is normally compiled into,

```
get.a lit.1 add put.a
```

It can be replaced by generating the native code "Inc.a", "Dec.a".

The last two rules come from the observation that in the "if" expression, the conditional becomes:

```
get.a lit.0 eq jf
```

The sequence "lit.0 eq if" can be replaced by "jt". So the conditional becomes,

```
get.a jt
```

The rule 4.1 and 4.2 used this fact.

The code generator fragment for doing "inc/dec" is as follows. "genput" is the main function. It is called when the operator "put.a" is encountered at the beginning of the expression (in N-code) (put.a (add get.a lit.1)). "isIncDec" checks whether the expression is in the increment/decrement expression. If it is then generate the native instruction, otherwise generate the normal unary-op code (in "genuop").

```
(def genuop (op arg e) ()
   (do
   (eval e)
   (outa op arg)))

(def isIncDec (op arg e) ()
   (do
   (if (isATOM e) 0
   (if (!= (head e) (mkATOM op 0)) 0
   (if (!= (arg2 e) (mkATOM xGET arg)) 0
   (if (!= (arg3 e) (mkATOM xLIT 1)) 0
   1))))))

(def genput (op arg e) ()
   (if (isIncDec xADD arg e)
      (outa icInc arg)
   (if (isIncDec xSUB arg e)
      (outa icDec arg)
   ; else
   (genuop op arg e))))
```

The code fragment for transforming (!= p 0) to p is as follows (at the function "gencall"), similar to the generating the primitives. Where yNe is the index to the user-defined function (def != ...). If the call is "!=" and the second argument is "lit.0" then generate the native code, otherwise generate the normal code (as function call).

```
(if (= idx yNe)
   (if (isOpArg (arg2 e) xLIT 0)     ; (!= p 0) -> p
      (eval (head e))
      ; else
      (genop icNe 0 e))
```

The code fragment for "short-cut" jump is straight forward and we will not elaborate any further. The code fragment for performing reduction in "if" expression is as follows. Where "genif2" is the normal "if" generation, "iseq0" checks the expression of the form (= a 0).

```
(def genif e (e1 e2 e3 ads)
   (if (iseq0 (head e))              ; (= a 0)
      (do
      (set e1 (arg2 (head e)))       ; e1 is a
      (set e2 (arg2 e))
      (set e3 (arg3 e))
      (if e3                         ; (if (= a 0) x y) -> (if a y x)
         (genif2 (cons e1 (cons e3 (cons e2 NIL))))
         ; else
         (do                         ; (if (= a 0) x) -> if a skip x
         (eval e1)                   ; a
         (outa icJt 0)
         (set ads (- XP 1))
         (eval e2)                   ; x
         (patch ads (- XP ads)))))
   ; else
   (genif2 e)))                      ; normal if
```

The code generator, "gen5.txt" is modified with these rules. The result is the code generator which generates primitives: !=, <=, >=, and, or, not and with the above improvement, inc, dec, short-cut jump, reduce != and improve conditional (if (= a 0)..). Let this code generator be "gen6.txt".

The step to run benchmarks of this new code generator is:

1 Compile the code generator.

```
c:>nvm nut.obj < gen6.txt > gen6.obj
```

Table 9.6 The profile of the benchmarks. Comparing the primitive and the codegen. (1) is codegen/primitive

	Compiler		Message passing		
	instr. x1000	cycle x1000	instr	cycle (system:user)	cycle/ switch
primitive	6855	28987	40688	206450 (44468:161982)	222
codegen	6412	27483	38599	199088 (44036:155052)	220
(1) %	93.5	94.8	94.8	96.4	

2 Use the new code generator to generate the executable object of the compiler.

```
c:>nvm gen6.obj < nut.obj > nutg.obj
```

3 Use the new compiler to compile the benchmark.

```
c:>sx nutg.obj < nut.txt
```

Similarly for the operating system benchmark, do the following steps.

1 Generate the executable Nos.

```
c:>nvm gen6.obj < nos2.obj > nos2g.obj
```

2 Run the new Nos2.

```
c:>noss nos2g.obj
```

The result is shown in Table 9.6.

The “gen6.txt” code generation is built on top of the “gen5.txt” (primitive) code generation so the performance improvement is considered relative to “gen5.txt” (a kind of further improvement of “gen5.txt”). The improvement in code generation further reduces the cycle by 5% in compiler benchmark and 4% in message passing benchmark. It reduces the task switching cycle by only 1%. Inspecting the profile of the compiler benchmark revealed that the “inc” is used 123 times, the instructions in the sequence that it replaced, consists of “get.a lit.1 add put.a” are reduced by the same amount. The profile of message passing benchmark is similar, but the “short-cut jump” which is not much in effect in the

compiler benchmark, is quite effective here, the number of “jmp” is reduced by 50% (from 800 to 401).

9.5 Instruction set level

The next level is the level of the instruction set. Although the instruction set is considered as “given” in any real computer system, to understand why an instruction set is designed that way, one should try to experiment with the effect of instruction set design [CHO03]. Observing the profile of the baseline benchmarks, two facts emerge.

1 The compiler benchmark is dominated by the execution of the function “str=”.
2 Both benchmarks, the instructions to access data structure, mostly “ldx” and “stx” are used often, 370/8585 = 4.3% in compiler benchmark, 2821/43580 = 6.5% in message passing benchmark.

By introducing new instructions, these functions can be much faster. Implementing new instructions in the processor can be accomplished by writing new microprograms for those instructions in the Sx processor. Tools are available to create and execute new instructions in the Sx processor.

Let us start with the fact 1, the string comparison function. Here is the “str=” function from the Nut-compiler, “nut.txt”.

```
; test string equal
(def str= (s1 s2) (flag i c1 c2)  [lib 28]
   (do
   (set flag 1)
   (set i 0)
   (while flag
      (do
      (set c1 (vec s1 i))
      (set c2 (vec s2 i))
      (if (!= c1 c2) (set flag 0)
      (if (= c1 0)   (set flag 0)
      (if (= c2 0)   (set flag 0))))
      (set i (+ i 1))))
   (and (= c1 0) (= c2 0))))
```

We should design the new instruction for the *inner* loop of this function. The inner loop does fetching two characters and compares them, at the same time it must checks the termination of strings. We do not try to do iteration within an instruction because it causes a long multiple cycles which is not desirable, especially at the microarchitecture level. It can cause unpredictable delay. But we try to do as much as possible in an instruction, so we will include the increment of index (set i (+ i 1)) in the instruction. Let the new instruction for string comparison be "string equal and increment" (seqi). Here is its pseudo code. Let *p1* and *p2* be two pointers to strings, *s1* and *s2* consecutively.

```
seqi p1 p2
  c1 = *p1                fetch a character
  if c1 == 0 ret 0        s1 terminate
  c2 = *p2                fetch a character
  if c2 == 0 ret 0        s2 terminate
  if c1 != c2 ret 0       if not equal ret false
  p1++                    increment both pointers
  p2++
  ret 1                   ret true
```

"seqi" fetches and compares two characters from two pointers to strings. It returns true if they are equal, otherwise it returns false, including when either pointers pointed to a terminal character. If it returns true, it also increment both pointers. This is very nice abstraction of the inner loop of the "str=" function. It does not do iteration and it does almost everything else (as much as possible). With the primitive "seqi" as a built-in operator, the "str=" function can be written as follows.

```
(def str= (s1 s2) ()
   (do
   (while (seqi s1 s2) (nop))                ; loop until false
   (and (= (vec s1 0) 0) (= (vec s2 0) 0))))
```

The instruction "seqi" has two arguments. This will introduce a new instruction format to the S-code instruction set. The two-address format is as follows.

16	8	8
a2	a1	op

Therefore some new signals in the data path must be added to decode this instruction format. Let they be *y.a1* and *y.a2* to feed the argument *a1* and *a2* to the y-mux in the data path.

The next step, we must write the microprogram for the "seqi" instruction. Here is its microprogram.

```
<seqi>                                ;  seqi.a1.a2
  sp+1, pc+1
  ts->mW(sp)                          ;  push ts
  alu(fp-a1)->tbus, mR(tbus)->nx      ;  read p1
  mR(nx)->ts->ff                      ;  read *p1,save
  alu(ts=0) ifT fetch                 ;  ret 0
  alu(fp-a2)->tbus, mR(tbus)->nx      ;  read p2
  mR(nx)->ts                          ;  read *p2
  alu(ts=0) ifT fetch                 ;  ret 0
  alu(ts=ff)->ts ifF fetch            ;  ret 0
  alu(nx+1)->ts                       ;  p2++
  alu(fp-a2)->tbus, ts->mW(tbus)      ;  update p2
  alu(fp-a1)->tbus, mR(tbus)->nx      ;  read p1
  alu(nx+1)->ts->ff                   ;  p1++
  alu(fp-a1)->tbus, ts->mW(tbus)      ;  update p1
  alu(nx!=ff)->ts fetch               ;  1->ts
```

The last line of microprogram is a good trick. To create a "*true*" value, we use the fact that "*nx*" is not equal to "*ff*" (we do not have any "*true*" value as a constant in the data path). This fact follows from the assignment in the line "*alu(nx+1)->ts->ff*", therefore the "*nx*" and "*ff*" will be definitely not the same. To do the function "not-equal" in the arithmetic logic unit, a signal "*alu.ne*" is added to the signal list in the microprogram specification, "mspec.txt". The timing of the "seqi" instruction is as follows. If it returns false, it takes one of the following cycle 6 or 9 or 10 cycles; if it returns true, it takes 16 cycles.

Now, let us turn our attention to the fact 2, accessing data structure. By inspecting the listing of the code (S-code), the data structure access is mostly in this form: (vec a index), (setv a index value), where a is the base address, index is mostly a constant. This observation suggests immediately the following instructions (also in two-argument format):

```
ldxv.a.x        TS = M[ M[fp-a] + x]
stxv.a.x        M[ M[fp-a] + x ] = TS
```

This is a kind of index addressing mode where the index is a constant. They are useful to access a structured data type such as a record. This kind of data is used very often in the benchmark programs.

Here are their microprograms.

```
<ldxv>   [micro 329]
  sp+1
  ts->mW(sp)              ;  push ts
  alu(fp-a1)->tbus, mR(tbus)->ts
  alu(ts+a2)->tbus, mR(tbus)->ts fetch

<stxv>   [micro 335]
  alu(fp-a1)->tbus, mR(tbus)->nx
  alu(nx+a2)->tbus, ts->mW(tbus)
  mR(sp)->ts          ;  pop ts
  sp-1, pc+1, fetch
```

Now the sequence of instruction for (vec a 0) and (setv a 3 44) which are normally compiled into:

```
get.a lit.0 ldx
```

and

```
get.a lit.3 lit.4 stx
```

will become a shorter sequence.

```
ldx.a.0
```

and

```
lit.44 stxv.a.3
```

“ldxv” takes slightly more cycle than “ldx” (“ldxv” 5, “ldx” 4 cycles). Surprisingly “stxv” takes less cycle than “stx” (“stxv” 5, “stx” 7 cycles). This is

because the “stxv” has its two arguments ready in the instruction; it does not have to pop the evaluation stack as much as “stx”.

Finally, the sequence “eq, jf” is founded often; it will be replaced by a new instruction “jne”. In fact, many of this “eq, jf” have already been optimised away when transforming (if (= a 0) x y) to eliminate the (= a 0) but the remaining is still significant.

```
<jne>   [micro 339]
  mR(sp)->ff
  sp-1
  alu(ts=ff), ifT j3
<jump>                    ; jump
  pc+arg, mR(sp)->ts      ; pop ts
  sp-1, fetch
<j3>                      ; don't jump
  pc+1, mR(sp)->ts        ; pop ts
  sp-1, fetch
```

How to generate microprogram

Now we have four new instructions: seqi, ldxv, stxv, jne (the inc, dec instructions have already been implemented in the previous chapter). These four instructions must be installed into the processor simulator. Assume all the additional signal definition and the appropriated microprogram are added into the input file, the microprogram specification, “mspect.txt”.

It is two steps to generate a microprogram. First, generate a microprogram bit-image. A microprogram bit-image is the raw data to be loaded into the microprogram control ROM. Second, to efficiently run the processor simulator, this bit-image is converted to an event list. It is 10 times faster when performing simulation using an event list.

1 Generate the bit-image file, mpgm.txt

```
c:>mgen > mpgm.txt
```

mgen implicitly takes the input file “mspec.txt”. mgen outputs another file “mspec.h” as a header file of the signal definition.

2 Then generate the header file to be include in compiling the processor generator.

```
c:>sxgen
```

The `sxgen` reads input from two files “mpgm.txt” and “mspec.h” then it outputs one file “sxbit.h” which includes all signal definitions and the event list data structure to be used by the processor simulator.

The processor simulator, `sx`, has a small modification to include the new events and the order of execution of the events; two new functions to decode the IR fields (see the Sx processor simulator listing in the appendix G).

3 Recompile the Sx processor simulator.

```
c:>make sx
```

Similarly for the Noss simulator, include this “sxbit.h” and recompile.

```
c:>make noss
```

Code generation for new instructions

A new code generator is written on top of the previous code generator, “gen6.txt”. It becomes “gen7.txt”. This code generator accepts the primitive “seqi” and also generates code using “ldxv”, “stxv” and “jne” as applicable. Let the new “str=” be included into a modified Nut-compiler (we do not touch anything else in “nut.txt”), named “nut5.txt”. The steps of work to perform statistic collection for this final benchmark are.

1 Compile this new code generator, “gen7.txt”.

```
c:>nvm nut.obj < gen7.txt > gen7.obj
```

2 Compile “nut5.txt” to N-object.

```
c:>nvm nut.obj < nut5.txt > nut5.obj
```

3 Use “gen7.obj” code generator to generate code from this modified compiler.

```
c:>nvm gen7.obj < nut5.obj > nut5x.obj
```

4 Run this new compiler (that includes the extended instructions) with the updated simulator to compile the compiler benchmark. The "x" suffix in the final object code signified the extended instruction.

```
c:>sx nut5x.obj < nut.txt
```

The message passing benchmark is similar.

1 Use "gen7.obj" to generate code.

```
c:>nvm gen7.obj < nos2.obj > nos2x.obj
```

2 Run it under the updated noss.

```
c:>noss nos2x.obj
```

The result is shown in Table 9.7.

Table 9.7 The profile of the benchmarks. Comparing the baseline, the codegen, and the extend. (1) is extend/codegen (2) is extend/base

	Compiler		Message passing		
	instr. x1000	cycle x1000	instr	cycle (system:user)	cycle/ switch
base	8585	38033	43580	220915 (49274:171641)	246
codegen	6412	27483	38599	199088 (44036:155052)	220
extend	3777	17922	32559	171637 (37791:133486)	189
(1) %	58.9	65.2	84.4	86.2	
(2) %	44.0	47.1	74.7	77.7	

In terms of cycle, the new instruction set (labeled "extend" in the table) further reduce (relative to codegen) for the compiler benchmark, 35%, for the message passing benchmark, 14%. This shows that the "seqi" is very effective as it is designed to speedup the "str=" in the compiler. As a whole, comparing the

number of cycle from this new “extend” optimisation with the baseline, the result shows that for the compiler benchmark, the speedup is more than twice the baseline, 53%, for the message passing, the speedup is 23%. It is worth noticing that for the message passing benchmark, the macro expansion is more effective, the speedup is 46%. The fastest task switching takes only 114 cycles.

The table below shows the summary of the speedup figure of each optimisation method. The chart from this table is shown in Fig. 9.1.

Table 9.8. The summary of speedup of each optimisation method in terms of cycle

	compiler	message passing
base	0.0	0.0
macro	30.5	45.4
primitive	23.8	6.5
codegen	27.7 (5.2)	9.0 (3.6)
extend	52.9 (34.8)	22.3 (13.8)

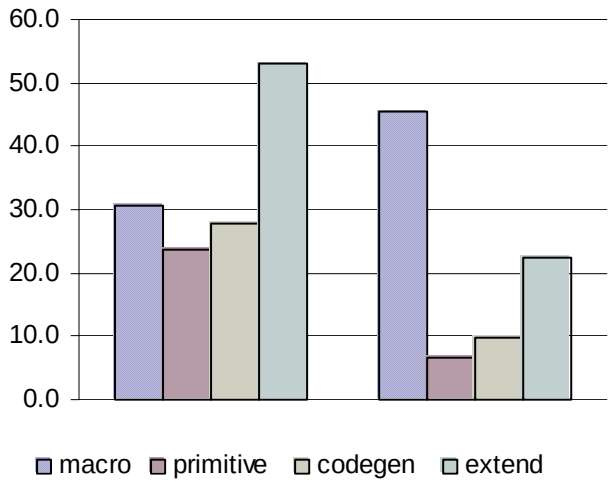

Figure 9.1 Chart of speedup (%) of each optimisation method in terms of cycle

The speedup is calculated from (1-(A/B)) x 100, where A is a method, B is the baseline. Speedup means how much faster the system A compared to the system B. If A is 20% faster than B, A completes the task in 20% less cycle than B. The figure in parentheses show the incremental speedup, for example, in the compiler benchmark, the method "codegen" is 5.2% relatively faster than the method "primitive", although it is 27.7% faster than the baseline. This shows the speedup factor from the incremental change of the method (the "codegen" method implemented many techniques *in addition to* the method "primitive").

9.6 Microarchitecture level

The chapter 7 has developed a processor that retains the same instruction set, S-code, but has 30% faster data path. The improvement at this level comes from the *architectural* design and the advancement in fabrication technology. These two factors are interrelated. The Sx2 processor is faster due to the *stack frame caching*, the use of fast registers to provide access to local variables instead of accessing them from the memory. The *parallelism* or the concurrent use of units in data path also provides performance enhancement. The use of separate unit for updating the stack pointer in Sx2 is an example of this case. Many other standard methods such as pipelining, using multiple functional units, *instruction level parallelism* are extensively discussed in many computer architecture textbooks [HEN03] [PAT98] [STO93].

9.7 Summary

In this chapter we have shown a wide range of performance improvement methods. The techniques are applicable at every level, from the top level (at the application software), down to the hardware level (at the level of data path and microprogram). At the highest abstraction level, it is a well-known fact that, the change at an algorithmic level will have the greatest impact on performance. We observe data that support this hypothesis. In the compiler benchmark profile, it is clear that the access to symbol table is the performance bottleneck. The symbol table requires a lot of string comparison. If however, the data structure is changed to a hash table, then the number of string comparison will be reduced dramatically. Our symbol table executes most accessed functions in O(n) where n is the number of the entry because it used sequential search. This will change to a constant access time, O(1), with a hash table (see Exercise 9.1).

The macro expansion is studied and implemented. It has a high impact on performance without having to change the underlying programs much (such as the code generation and the instruction set). Another change at the language level is to introduce more operators. Our experiment shows that this is also effective although not as much as macro expansion. However it is not effective in the message passing benchmark because these new operators are not used very often. This shows that it is not easy to gain performance in general by this method. It is sensitive to the type of applications.

The next level is the code generation level. The quality of code can be improved. Most methods at this level are classified as *code optimisation* in the literature of compiler. We choose to perform a number of significant code improvements. The result shows that the performance gain is not large but it is very logical method. The rule for replacing some sequence with a shorter sequence to improve performance is numerous (in fact, it is combinatorial even!). One can almost invent a new rule by inspection the output code. However, how often that sequence will be used is not always easy to predict.

The instruction set level is interesting. Inventing new instructions is not practical in a real computer system where hardware is *given*. However, our study shows that a huge performance gain is possible. The “string equal and increment” instruction is an extreme example. It is more than twice as fast as the code without this instruction in the compiler benchmark. However, it has a limit application to string comparison. This is why the multimedia extension of an instruction set is so important for modern processors.

9.8 Further reading

In general, a programming language should not be designed with paramount of efficiency in mind. The high level language should reflect the efficiency of human programmers, at the cognitive level. It should reduce the burden of human programmers. During early development of computer systems many components were being developed: the hardware, the compiler, the operating system, the user interface etc. The designers and the developers faced the difficulty of not having the machine fast enough to run the intended applications. This must be the nature of computer technology that human can imagine new kind of application beyond what a computer system can support at the time. So, in all history of development of computer systems, efficiency is always a

paramount factor. The C language [KER78] reflects this fact. Pascal [WIR71] also reflects this concern. Computer language design is still an open, endless quest for a tool of thought to invent the next generation machines. The history of computer development can be read in IEEE Annals of the history of computing.

For code optimisation, many compiler related textbooks are excellent source of information [AHO86] [FRA95] [LOU97]. Although it seems that code optimisation is used for code generation, it application can be wider range. For example, it is also applicable at the *meta* level programming, to build a virtual machine (or environment) [CHO98], to prototype a new computer system.

References

[AHO86] Aho, A., Sethi, R. and Ullman, J. Compilers: Principles, techniques, and tools, Addison-Wesley, 1986.

[CHO98] Chongstitvatana, P. A multi-tasking environment for real-time control. Final report, Faculty of Engineering, Chulalongkorn university, research project number 132-MRD-2537, 1998. Also available on-line at http: //www.cp.eng.chula.ac.th/faculty/pjw/r1/

[CHO03] Chongstitvatana, P., "The Art of Instruction Set Design", Electrical Engineering Conference, Thailand, 2003.

[COR01] Cormen, T., Leiserson, C., Rivest, R., and Stein, C., Introduction to algorithms, 2nd ed., MIT Press, 2001.

[FRA95] Fraser, C. and Hanson, D. A retargetable C compiler: design and implementation, Benjamin/Cummings Pub., 1995.

[HEN03] Hennessy, J., and Patterson, D., Computer Architecture: a quantitative approach, 3rd ed. Morgan Kaufmann, 2003.

[KER78] Kernighan, B., and Ritchie, D., The C programming language, Prentice-Hall, 1978.

[KLI05] Klienberg, J., and Tardos, E., Algorithm Design, Addison Wesley, 2005.

[LOU97] Louden, K. Compiler construction: Principles and practice, Inter. Thompson Pub., 1997.

[PAT98] Patterson, D., and Hennessy, J., Computer Organization and Design: the hardware/software interface, 2nd ed. Morgan Kaufmann, 1998.

[PRA01] Prasitjutrakul, S., Analysis and design of algorithms, NECTEC, 2001. (in Thai).

[SEB04] Sebesta, R. Concepts of programming languages, 6th ed. Pearson/Addison-Wesley, 2004.

[STO93] Stone, H., High performance Computer architecture, McGraw-Hill, 1993.

[WIR71] Wirth, N., "The programming language Pascal", Acta Informatica, 1(1):35-63, 1971.

Exercises

9.1 Do some simple optimization. At the level of algorithm, change the symbol table access to use hash table. Run the compiler benchmark with this new compiler. Collect the profile and discuss the result.

9.2 Do some language extension. In the Nut-compiler and the code generator, there are frequent uses of multiway branch. To do multiway branch, nested-if is used. This is very flexible but it is inefficient to do sequential testing on the conditions. Also, the syntax for deep nested-if is quite cumbersome, especially the closing parenthesis. The number of right parenthesis is equal to the depth of nested-if.

```
(if (= op xADD) (doadd)
(if (= op xSUB) (dosub)
...
; else
(error "unknown op") ...))
```

To help improving the syntax in general case, a new control-flow operator is needed. The expression (cond....) in LISP is a good example. In Nut, we want to do:

```
(switch
  (cond1) (action1)
  (cond2) (action2) ...
  true (default action))
```

This syntax will reduce the number of the closing right parenthesis to one.

To improve the efficiency, if the condition is testing the equality of a variable with many constants, then we can do similar to “switch, case” in C or Pascal. The implementation can use a jump-table indexed by the value of that variable.

```
(case op
  (label xADD (doadd))
  (label xSUB (dosub))
  ...
  (else (default action)))
```

The N-code for “case” will contain a jump-table of the association list of (label jump-to) which is implemented as an array of cells. To conform to the existing N-code only the head can be an atom, therefore the head is a label instruction (a new instruction), the tail is a pointer to the action body.

In N-code, assume op is local, len is length of the jump-table, and the last entry is an instruction to terminate the table. The “case” construct is compiled to:

```
(case.op
  ; jump table
  (label.xADD L1)
  (label.xSUB L2)
  ...
  (else.0 Ln))
  ; actions
  <L1> (action 1)
  <L2> (action 2)
  ...
  <Ln> (default action)
```

If the label is sorted according to the key (constants) then the jump table can be a binary search table. The length of jump-table must be known.

```
(case2.op lit.len
  ; jump table, label is sorted by its arg
  [xADD L1
   xSUB L2
  ...
   NIL Ln])
  ; actions
  <L1> (action 1)
  <L2> (action 2)
  ...
  <Ln> (default action)
```

Another way which is faster is to use label to index into the jump table directly. The size of jump table is equal to the range of index. The label instruction is not necessary. The range of index must be known.

```
(case3.op lit.lo lit.hi
  ; jump table, label is sorted by its arg
  [L1 L2 ... Ln])
  ; actions
  <L1> (action 1)
  <L2> (action 2)
  ...
  <Ln> (default action)
```

For the second kind and third kind of "case", the jump-table itself does not conform the format of N-code. To implement it, it is not really a list; it is an array (which is a data). To have an array in the code segment, an additional representation must be designed.

Implement a multiway branch. Run the benchmark and measure its effectiveness.

9.3 Write a code generator for S2, the register-based processor. Compile all the benchmarks to target this instruction set. Run and collect the performance statistic. Compare it with Sx processor. Discuss the result.

9.4 One can always argue that some inefficiency comes from the quality of a compiler. To prove or disprove this belief, we can write a segment of program in machine language then compare the execution time with the

one produced by a compiler. Write the string comparison function in S-code by hand. Try to optimise it by speed. Compare the running time with the one produced by Nut-compiler.

9.5 Invent new instructions by combining two instructions into a new one. Produce a log file of the frequency of pair of instructions and select five most frequently used pairs. Write their microprogram. Integrate these new instructions into the Sx processor simulator. Modify the code generator to output these new instructions. Run the benchmarks and discuss the result. How much do you expect the improvement will be?

9.6 Performance measurement is sensitive to benchmark programs. Write a new set of benchmark programs, for example, image processing. Measure the performance of our system on this new benchmark. What is the difference and why?