Advanced Compiler Optimization for CalmRISC8 Low-End Embedded Processor

Dae-Hwan Kim

MCU Team, Samsung Electronics, San #24, Nongseo-Ri, Kiheung-Eup,
Yongin-City, Kyungki-Do 449-900, KOREA
dhtail@samsung.co.kr

Abstract. CalmRISC8 is an 8-bit embedded processor, in which architectural considerations for compiler are ignored to reduce power consumption. To overcome these constraints, new techniques are presented at an intermediate code level, an assembly level, and a link-time level. Techniques include register allocation, integer promotion elimination, extensive use of library functions, instruction scheduling for bank collects, and various optimizations at link-time. Experimental results show that 56.7 % reduction in code size can be achieved.

1. Introduction

In recent years, microcontrollers and DSPs are increasingly being embedded into many electrical products such as a fax modem and a cellular telephone. A common trend found in embedded systems is to integrate a microcontroller, a ROM, a RAM, and an ASIC on a single IC. In traditional compilers, especially for general purpose processors, optimization of generating fast executable code is mainly focused on. However in some embedded processors, the compactness of code is at least as important as execution speed. As the application software often exists in an on-chip ROM, so large code size increases ROM size and eventually the cost of system. On the other hand, as the complexity of embedded software is relatively small, the execution time improvement is often marginal. Further, many embedded processors have architecture constraints to generate a compact code. Such examples are relatively small number of registers and limited instruction set, and irregular data paths. Thus code reduction by traditional techniques is not satisfactory, and to get a dense code on such architectures, new optimization techniques of overcoming architecture constraints should be applied to.

CalmRISC8 [1] is an 8-bit embedded processor developed by SAMSUNG Electronics, and its main goal is to reduce power consumption. It operates on 3-volts of power and requires 30-70 microamperes per megahertz. To accomplish its low power consumption goal, architectural preferences by compiler are less considered or even ignored. The CalmRISC8 register set is organized as 4 banks with 4 registers in each bank to minimize the number of bits that the register specification consumes in instruction format. No instructions support operations on registers across different banks. This bank structure of registers greatly reduces the register allocation performance and causes a bank switching overhead to use registers in different banks.

The limitation in addressing mode is another example of an architectural constraint that makes optimization difficult. In most cases, the memory operand is accessed by specifying the address in designated registers named 'index registers'. The register indirect addressing with a general purpose register (GPR) is not available.

The operations on word (a 16-bit length) on an 8-bit machine are also an obstacle. This becomes more overhead in CalmRISC8, because the registers in one bank often run short, so bank switching is required. CalmRISC8 has the 16-bit address line. The data memory of CalmRISC8 is organized as 256 pages of 256 bytes each. The high-order part of 16-bit address is page number and lower-order is page offset. In some cases, the sequential memory access is performed in a same page, thus, memory access can be optimized by deleting the latter page set instruction. Useless integer promotion from ANSI C makes many word operations also. Integer promotion means the type conversion from character to integer types when operands are character variables. When operands are character types, much of conversion can be eliminated.

Many of techniques in this paper are specific to CalmRISC8. However, basic ideas can be adapted to most DSP processors which have the multiple register bank and memory bank problem. Some of the issues occur also in non-DSP processors. For example, a VLIW may have the multiple register and memory banks.

The rest of this paper is organized as follows. Section 2 describes CalmRISC8 architecture. In section 3, overall structure of optimization is explained. In section 4, 5, 6 this paper presents intermediate, assembly, and link-time level solutions for architecture limitations in detail respectively. Experimental results are shown in section 7. This paper concludes in section 8 with some comments.

2. CalmRISC8 Architecture

Fig. 1 shows the CalmRISC8 architecture. CalmRISC8 is an 8-bit embedded processor in which register and a data path are 8 bits long. The main characteristics of CalmRISC8 can be summarized as follows.

- CalmRISC8 follows the Harvard architecture in which program and data space is separated.
- Program memory is organized as 256 pages of 4K words for each page and data memory is 256 pages of 256 bytes each.
- For data memory addressing, index register pairs (IDH and IDL) should be used or absolute address should be specified in the instruction. CalmRISC8 has 16-bit data memory address line. The upper 8 bits of the data address that should be reside in IDH point to a specific page and the lower 8 bits specify the offset address of the page.
- It has two banks of index registers, each of which has one IDH and IDL0, IDL1.
- It has four banks of four general purpose registers in each bank.
- To change a bank of general purpose registers or index registers demands a special instruction. Current bank information is in SR0 register. The instruction for changing bank is one of 'BANK #num' or 'ORSR0 #imm' or 'ANDSR0 #imm'.

- It is a non-load/store architecture. Some operators can use memory operand as second operand. This kind of operators are AND, OR, XOR, ADD, SUB, CMP, LD.
- There are two types of branches including calls. One is a short branch (1 word) which branches into the address of the code memory in the same page, the other is a long branch (2 word) in which a target address can be in other pages.

Fig. 1. CalmRISC8 Architecture

The addressing modes of CalmRISC8 can be demonstrated by 'LD' instruction. The page number of the address should be in an IDH register and offset can be in IDL0 or IDL1 or absolutely specified in the instruction. Thus, following LD instructions are possible for memory access. The displacement can be combined with index registers, and the displacement should be within 5-bit range.

```
LD    R1, @ID0  // address is 16 bits of (IDH:IDL0)
LD    R1, @ID1  // address is in (IDH:IDL1)
LD    R1, @[ID0+16]! // address is (IDH:IDL0)+16
LD    R1, @7A   // address is (IDH:7AH)
```

Due to the architecture constrains for compiler, CalmRISC8 compiler generates relatively long code without optimizations. Fig. 2 shows assembly code before optimization for the statement of 'a[i] = 0;', where 'a' is the global scope array and 'i' and 'a' are of character types. To compute the data memory address, first the value of 'i' is loaded into R3 from data memory. Then the value of 'i' is extended to integer by 'CALL __cvci_r3', which is the library function and it takes the register R3 in bank 0 and extend the sign bit into the R3 in bank 1. It is then added to the base address of 'a'. The '#<_a' represents the high part of 16 bit address of 'a' in data memory and resolved at link-time. The '#>_a' does the lower part. The 'ADD' instruction permits the immediate value as the second operand, but 'ADC' does not, so in 'ADC' in line (8) the second operand is in register. Then, the address of 'a[i]' is moved into IDH and IDL1 that are the index registers in line (13) and (15). The 'ID1' means combination of IDH and IDL1. The IDH and IDL0 are used as the stack pointer. Thus to keep

the value of IDH which is written at line (13) of the 'LD IDH, R3', the IDH is saved by 'PUSH' instruction and later restored by 'POP' instruction.

LD R3,@[ID0+(-1+1)]!:_i
CALL __cvc_i_r3
BANK #1
LD R2,#<_a
BANK #0
ADD R3,#>_a
BANK #1
ADC R3,R2
BANK #0
LD R2,#0
PUSH IDH
BANK #1
LD IDH,R3
BANK #0
LD IDL1,R3
LD @[ID1]!,R2
POP IDH

Fig. 2. Generated code example before optimization

3. Overall Structure of Optimization

To overcome architectural constrains for compiler, new techniques targeted for the CalmRISC8 are contrived. Optimizer runs at intermediate code, assembly code, and link-time cooperatively. The newly designed register allocation is applied for small set of registers and by using extensive assembly library functions like the line (2) in Fig. 2, the effect of expanding instruction set including addressing modes is achieved. The word operations are reduced byte operations by eliminating integer promotion. To reduce overhead of bank switching, instructions are reordered to gather same bank operations. At link-time, the cost of global variable access is minimized, word operations are reduced to byte operations again for functions whose stacks are allocated in a single page, and the 2-word branch instructions are changed into 1-word instructions when the target label is in the same page with the branch label.

General structure of optimization is shown in Fig. 3. The intermediate optimizer performs optimizations including classical optimizations [2] on DAG form of intermediate code. Implemented techniques are common subexpression elimination, copy propagation, constant propagation/folding, dead code elimination, peephole optimization, and optimal instruction ordering. In addition, it passes optimizing information to an assembly and a link optimizer. Some example of information is the condition and code block that can be deleted at link-time. All optimizations in an intermediate level are implemented in Local C Compiler (LCC) [3]

An assembly optimizer does the classical optimizations, target specific optimizations, procedural abstraction, and instruction scheduling for bank collection. Classical optimizations are performed on assembly level again, because some code redundancy

is caused from machine resources and architecture limitation. An assembly optimizer analyzes information from an intermediate optimizer, changes it, and generates its own optimization information for the link-time optimizer. It takes the assembly code from intermediate optimizer and transforms code optimally and passes assembly code with pseudo code to assembler.

A link-time optimizer is a part of linker. CalmRISC8 linker plays a role of a standard linking job. Further, some code optimizations are performed. The code change at link-time is only deleting. It affects main job of linker such as external symbol reference and resolution. Deleting the code at link-time is a hard part, because this requires reallocation of code address. At some transformations like stack optimization, linker manipulates and constructs information from assembler actively in itself.

Fig. 3. Overall optimization structure

4. Intermediate Optimization

In CalmRISC8, the size of word is 16 bits and byte is 8 bits. Like other 8-bit embedded compilers, the size of character is 8 bits and short and integer types have 16-bit size. CalmRISC8 compiler allocates one register in bank 0 for character and does pair registers of the same number from bank 0 and bank 1 to the word data that demands 2 bytes. It reduces the number of library functions, compared with intra-bank register allocation, which allocates 2 registers in the same bank.

4.1 Flow Analysis OPT Register Allocation

CalmRISC8 has four general purpose registers in each bank of four, and operations on registers between different banks are not permitted. Therefore, it can not have better performance than that of four registers of no bank structure. The known dominant paradigm of register allocation is a graph coloring [4], [5], [6]. However, graph

coloring does not result in good performance with small register sets such as CalmRISC8.

CalmRISC8 allocator takes the idea of the OPT page replacement algorithm in operating system. Contrary to O/S, compiler can analyze variable access sequence in most of cases, especially within a single basic block. When allocator runs out of registers, the register held by a variable that will be used at the furthest in the future will be replaced. Previously, the similar ideas are tried for local register allocation [7], [8]. However, they consume lots of spaces and time. Furthermore, they are local to a basic block.

Our method has two steps. At the first step, our OPT register allocation runs for each basic block. Before actually allocating registers, allocator constructs definition-use chains for each variable definition in an access order. Allocator tries to assign the same register to the same variable. At the second step, like the traditional data flow analysis, variable flow is analyzed for entire basic blocks. When the same variable has different registers from control flow, code is inserted to write variable content back into memory at the end of the basic block that has a variable definition. In the same way, loading operation from memory is inserted immediately before the use of that variable in descendant basic blocks.

To reduce the register pressure, temporary uses of word operations are split into two byte operations with the same register number for high-order and low-order part of operations. This reduces the number of required registers by halves, and obviates the bank-switching overhead. Word operations occur to some extent, even if user does not use much of word type variables. The address calculation and parameter passing cause some word operations.

```
LD R2, @i           // low part of i
BANK #1
LD R2, @ (i+1)      // high part of i
BANK #0
ADD R2, @j           // low part of j
BANK #1
ADC R2, @(j +1)      // high part of j
BANK #0
LD @k, R2            // low part of k
BANK #1
LD @(k+1), R2        // high part of k
BANK #0
```

(a) before split

```
LD R2, @i           // low part of i
ADD R2, @j           // low part of j  -(2)
LD @k, R2            // low part of k
LD R2, @(i +1)        // high part of i
ADC R2, @(j+1)         // high part of j  -(5)
LD @(k+1), R2         // high part of k
```

(b) after split

Fig. 4. Word operation split example. This is not perfect assembly code, but enough to illustrate the effect of idea.

Fig. 4 is the example of word operation split for the statement of 'k = i + j', where 'i', 'j', and 'k' are integer variables. In (a), values are first loaded into registers and the addition of word type is executed using 'ADD' and 'ADC' pair. After applying word operation split, code is shown in (b). The code 'LD' instruction does not affect carry information, so the instruction (5) uses the carry generated from instruction (2).

The intermediate expression tree can be split if any of word operations do not change the code correctness. Some operators on words can not be split. The shift of word by word is such an operator. The shift operator can not be transformed into two character shifts without additional code to keep the correctness. These operators are shift, type conversion, modular, and divide operator. The sequence of word operations that generate more than one carry can not be split also. The statement of ' $l = i + j + k$ ' can not be split, because that changes the carry information to be used. As an example, consider the code in Fig. 5. The line (6) should use the carry result of lower part of operation of ' $i+j$ ', that is the carry from line (2). However, if split, the (6) will use the carry of ' $(i+j)+k$ ', that is the carry from line (3). It is wrong.

```
LD R1, @i      // low part of i
ADD R1, @j      // low part of j -(2)
ADD R1, @k      // low part of k -(3)
LD @l, R1      // low part of l
LD R1, @(i+1)   // high part of i
ADC R1, @(j+1)  // high part of j -(6)
ADC R1, @(k+1)  // high part of k
LD @(l+1), R1   // high part of l
```

Fig. 5. Wrong word operation split example

4.2 Instruction Set Expansion by Library Functions

The CalmRISC8 has small number of instruction set, and many of them are primitive, so intermediate code corresponds to many assembly instructions. Furthermore, compiler generates many word operations. The word operations take a form of pattern. For example, word addition is composed of 'ADD' and 'ADC' pair. A deficiency of addressing mode is another factor to cause generating long code. To overcome above circumstances, making assembly library for code generation is tried, and it results in remarkable code reduction.

Table 1. Some of library functions

LIBRARY NAME	DESCRIPTION
__addW_rx_ry	Word (16 bits) addition
__cmpW_rx_ry	Word (16 bits) compare
__loadB_rx_ry	Register indirect byte (8 bits) load support. The 'rx' is the address of data memory and 'ry' is the register into which value is loaded
__storeB_rx_ry	Register indirect byte store
__loadW_rx_ry	Register indirect word load
__LshW_rx_ry	Left shift 'rx' by 'ry'. CalmRISC8 ISA only supports 1bit shift.
__loadB_storeB_rx_ry	The value of memory addressed by 'rx' is stored into the memory of address 'ry'
__add_loadB_rx_ry_rz	C language support. In most cases, used for array element referencing. The memory content whose address is 'rx' + 'ry' is load into 'rz'

Table 1 shows some assembly libraries that compiler frequently generates. There are about 50 kinds of functions. Some functions such as 'loadB_storeB_rx_ry' are the combination of other primitive library functions. In table 1, 'rx' and 'ry' can be a register 0, 1, 2, 3. As shown in Table 1, compiler generates like this:

```
CALL addW_r2_r3
```

for 16 bits addition, rather than:

```
ADD R2,R3
BANK #1
ADC R2,R3
BANK #0
```

To make assembly level analysis and optimizations easy, the intermediate optimizer generates information of reading and writing registers of the library as pseudo code. The format is that the first digit means bank number and the second represents register number as shown below.

```
.READ_REG 03,02,13,12
.WRITE_REG 03,13
CALL addW_r2_r3
```

4.3 Integer Promotion Elimination

In an 8-bit embedded processor application, most of the variables are 8 bits or character type. According to ANSI C, the character operands are widened into integer types for operations between character variables ('integer promotion'). When semantics can be preserved, this integer promotion can be eliminated. Additionally, integer operators are changed into character ones.

(a) before elimination (b) after elimination

Fig. 6. Integer promotion elimination example for the statement of “k = i + j “, when ‘i’, ‘j’, ‘k’ are character variables.

Fig. 6 shows the integer promotion elimination, when ‘i’, ‘j’, and ‘k’ are character variables. Without integer promotion elimination, compiler would generate the code for integer operations as shown in (a). First the ‘i’ and ‘j’ are converted into integers and the integer addition is executed, and then the result is written to ‘k’. However, in this example, the conversions from characters to integers and integer addition are of no use, because the result is written into character variable. In other words, excluding the possibility of overflow, the addition of character is equivalent to original code in

semantics. This elimination extends to argument passing. When the prototype is known, compiler compares the type of formal parameter with that of actual parameter. When the formal parameter is a character type, actual parameter of character type is not promoted into integer.

For each tree, one pass is in bottom up manner to check that the node can be converted by checking the kids' type. Take division operator of integer type as an example. Whether integer division can be converted into character division depends on the division operands. If both are in character boundary, it can be converted. However, if any of both should keep an integer value, it can not be changed. Thus for character variables 'i' and 'j', the integer division operator in 'i/2' can be converted into character division operator, because it is evident that 'i/2' has a character value. However, the divide in '(i + j)/2' can not be reduced because 'i+j' can have value of integer range.

A second pass is top-down approach to determine whether each node can be reduced. Basically, all promotions between characters including constants in character range are eliminated. However, some integer type operators keep promotions of character type from being eliminated. Some examples of these operators are assignment into integer variable, integer divide, and 16 bits address calculation. Thus, for example, in the statement of "`i_i = c_i + c_j;`" where '`c_i`' and '`c_j`' are character and '`i_i`' is a integer variable, the addition of '`c_i`' and '`c_j`' are promoted into integers, because that result is written into a integer variable.

4.4 Address Calculation Optimization

The word type address calculation is another barrier to overcome for generating dense code. The memory accesses except the local variable whose offset in stack is determined at compile-time, which takes the form of `@[ID0+offset]!`, demands word operations on address calculation. Such examples are global variable access and array indexing by non-constant expression. Relatively high percentage of the word addressing is address calculation of array element, and most of the array is byte addressable from the base address of array in an 8-bit application. In this case, array indexing can be optimized by the addition of byte offset and word base address rather than that of word offset and word base address.

LD	R3,@[ID0+(-1+1)]!	_i
CALL	\$_cvc_i_r3	
BANK	#1	;B1start
LD	R2,#<1	
BANK	#0	;B1end
LD	R2,#>1	
CALL	\$_lshi_r3r2	
BANK	#1	;B1start
LD	R2,#<_Gi	
BANK	#0	;B1end
LD	R2,#>_Gi	
CALL	\$_addW_r3r2	

(a) before optimization

LD	R3,@[ID0+(-1+1)]!	_i
ADD	R3,@[ID0+(-1+1)]!	_i
BANK	#1	;B1start
LD	R2,#<_Gi	
BANK	#0	;B1end
LD	R2,#>_Gi	
CALL	\$_addBW_r3r2	

(b) after optimization

Fig. 7. Address optimization

As an example, consider the computing the address of 'Gi[i]', assuming that the size of array 'Gi' is 100 bytes and the element type of 'Gi' is integer and 'i' is the character type variable. Without the address calculation optimization, the code is shown in Fig 7 (a). First, the value of 'i' is extended to word by the 'CALL \$_cvc_i_r3' instruction. Next, the index value is left shifted by 1-bit, because the size of integer is 2 bytes in CalmRISC8, then the offset is added to base address of 'Gi'. By moving the code of converting character to integer inside library function, code can be reduced. For this purpose, we create a new library function of 'addBW' library, that adds the second word value with the zero extended value of the first library argument. The body of 'addBW' is almost the same as 'addW'. Therefore, by sharing the same code, the total size of library functions is not increased much. In addition, to access the integer type array, the left shift of index variable by 1-bit can be reduced into an ADD operation. After the address optimization, the code is shown in (b).

5. Assembly Level Optimization

Two techniques are described here. One of well-known code compaction techniques is procedural abstraction that is more useful in low-end embedded processors and CalmRISC8 specific instruction scheduling.

5.1 Procedural Abstraction

Compiler generates the same code sequences that appear many times. In CalmRISC8, this becomes evident because it has a relatively small number of instructions and registers. Procedural abstraction makes each sequence as a separate function and changes repeated occurrence patterns into function calls [9].

Each assembly line excluding directives is mapped a symbol based on its opcode and operand types. Suffix tree [10] is built for these symbols. Thus, a whole assembly code is treated as one string. The non-terminal nodes that have at least 2 kids in tree are the candidates for a subroutine, because at least the pattern occurs 2 times. Next step is to evaluate the candidates based on the sequence occurrence count and the length of sequence. Current code size of totally repeated sequences is the multiplication of count by length. If abstraction is applied, the newly generated function code size will be addition of length and size of return instruction, and total size will be the addition of new function size and a multiplication of count by the size of call instruction. Procedural abstraction is applied for each pattern only when current total code size of patterns is greater than size that will be after abstraction.

5.2 Instruction Scheduling for Bank Operation Collects

One of main overheads of CalmRISC8 compiler is a need of bank switching. The current bank status is in SR0 (status register 0). The bit 3 and 4 have the GPR bank information and bit 2 has the index register bank information. To use other banks, bank specifying bits in SR0 should be changed by 'ANDSR0 #imm' or 'ORSR0 #imm'

or 'BANK #num' instructions. If we put together the same bank operations, many pairs of bank switching instruction become redundant. Our reorder technique is relatively simple compared with other instruction scheduling. The order of instructions in the same bank is not changed.

At an assembly level, the dependency between bank blocks that are the sequential instructions of having the same bank status is analyzed. In CalmRISC8, additional dependencies happen as well as normal data dependency. One of them is a carry dependency. The typical code form of 16-bit addition is like below.

```
ADD R0,R2
BANK #1
ADC R0,R2
BANK #0
```

Above code means ADC use the carry result of ADD. Therefore, ADC depends on ADD. No carry affecting instructions can locate within them and ADC should follow ADD immediately. Fig. 8 shows code before and after scheduling. In (a), the (5) can be moved after (1), because there is no dependency between them. After a move, (4) and (6) become useless, so those can be deleted as shown in (b).

LD R3,@[ID0+(-4+1+6)]! (1) jj(low)	LD R3,@[ID0+(4+1+6)]! (1) jj (low)
BANK #1 ;B1start (2)	LD R2,@[ID0+(-6+1+6)]! (5) kk(low)
LD R3,@[ID0+(-4+6)]! (3) jj(high)	BANK #1;B1start (2)
BANK #0;B1end (4)	LD R3,@[ID0+(-4+6)]! (3) jj(high)
LD R2,@[ID0+(-6+1+6)]! (5) kk(low)	BANK #0;B1end (4)
BANK #1;B1start (6)	BANK #1;B1start (6)
LD R2,@[ID0+(-6+6)]! (7) kk(high)	LD R2,@[ID0+(-6+6)]! (7) kk(high)
BANK #0 ;B1end (8)	BANK #0;B1end (8)
CALL __addW_r3r2 (9)	CALL __addW_r3r2 (9)

(a) code before scheduling

(b) code after scheduling

Fig. 8. Instruction scheduling code example of 'jj + kk'

This idea applies to the ID (index register) banks. ID bank 1 is used as a global variable pointer, whereas ID bank 0 is used for stack pointer of a function. Before a global variable access, the bank is switched into ID bank 1. After a global variable is accessed, compiler changes ID bank 1 back to ID bank 0. Like the GPR bank collect, the operations of global variables can be collected as well.

6. Link-Time Optimization

Only a few compilers have performed optimizations at link-time [11], [12], [13], because link-time optimizations need much analysis and the effects may not be great. Some embedded compilers have burdened the link-time information such as memory allocation information of variables on the users for efficient code generation. However, it is cumbersome to the users.

6.1 Global Variable Access Optimization

Many global variables have been used in an embedded processor application. Memory is allocated to global variables at link-time, so link-time is the best for global variable access optimization. To access global variables in CalmRISC8, two parts of address, which are the page number and offset, must be specified. Normally, the IDH (index high register) of ID Bank 1 specifies the page number of a global variable. The offset is often specified absolutely as shown (2) in Fig. 9 (a). Optimization occurs when two sequential global variables are allocated into the same page. In that case, the loading of page number to IDH of the second is the same as the first one. Therefore, it becomes redundant and can be deleted.

Fig. 9. Global variable access optimization example. This figure takes the example of 'j+k', when 'j' and 'k' are allocated into the same memory page. (a) shows pseudo code generated from intermediate optimization. (b) shows assembly information. In this code, sequential ID bank switching instructions are eliminated. (c) shows code after link.

The intermediate optimizer generates the name of global variable and marks the code block that can be deleted at link-time. Assembly optimizer analyzes the global variable access sequence throughout all basic blocks in a function. It specifies the link-time condition on that following code block, so it can be deleted. This mechanism is illustrated in Fig. 9. The (1) and (3) in (a) that is between GLOBAL_OPT BEGIN and GLOBAL_OPT END mark the block that can be deleted at link-time. The assembly optimizer generates variable access flow information like (1) and (2) in (b). For example, the pseudo code in (2) gives linker information that if the variable 'j' and 'k' are allocated in the same page memory, following marked block can be

deleted. The pseudo code of ‘.FLOW_GLOBAL’ lists all the global variables that can reach at that point from control flow and immediately following global variable name. The (c) shows code after link-time optimization. Linker deletes code and recalculates linking information such as code segment size and symbol linkage caused from code layout change.

6.2 Single-Page Stack Function Optimization

To overcome the paged memory architecture, in case that the stack, or activation record, of a function is allocated within a single page at all execution paths, many high-order parts of word operations can be deleted. In other words, word operations can be converted into byte operations. Furthermore, the save and restore instructions of index register pair to keep stack pointer, which are shown at line (11) and (17) in Fig. 2, can be eliminated. It is believed the embedded application use many global variables and relatively a few local variables, so the size of stack frame is relatively small. Thus in fact, many of functions will have stack frame within a single page, not covering many pages.

Fig. 10. Call graph and functions whose stack is allocated within a single page. In here, ‘F1’, ‘F4’, ‘F5’ are functions whose frames are allocated within a single page.

With these facts, like the global variable access optimization, this technique detects the functions of which stack are allocated within a single page from all paths in a call graph, and then deletes marked blocks at link-time. Intermediate optimizer generates function call information before ‘CALL’ instruction.

```
.fcall _foo
CALL  _foo
```

In addition, it generates stack size information for each function.

```
.fsize  frame_size (bytes)
```

Intermediate optimizer marks code blocks that can be deleted. This marking information takes the following form.

```
.STACK_OPT BEGIN
// code that can be deleted
.STACK_OPT END
```

Assembly optimizer constructs annotated call graph that has frame size information. Linker completes the call graph for entire program and marks single page allocatable functions from all call graph paths, and deletes marked blocks.

Take the call tree in Fig. 10 for an example, and assume the size of one page is 100 bytes for simplicity, and stack frame is allocated from address 0, and grows to high address. In this example, the function 'F1', 'F4', and 'F5' have frame in a single page by all paths. For example, the stack 'F5' will be in page 2 which path it follows. If 'F5' is reached from 'F1', 'F2', and 'F5' path, the stack of 'F5' begins from the address of 110 and the end of stack is 159. Thus, the stack is in page 2. From the other path, the stack of 'F5' is in page 2 also.

6.3 Branch Address Resolution.

In CalmRISC8, branch is not a PC-relative but absolute addressing, so at compile time, it is impossible to know whether target address is in the same page. Therefore, compiler emits the long branch instruction. At link-time, while resolving the address, the branch whose target is in the same page is changes into short one. If a branch is changed into short one, it causes code layout change, and affects other code address and other branch instructions. Therefore, this address resolution is iterated until no more code layout change occurs. This stage should be the last stage of link-time optimization, because other link-time optimizations can change code layout.

7. Experiments

The results of applying optimizations are shown in Table 2. On average, 56.7 % code reductions can be achieved. The code size after optimizations ranges from 36.8 % to 56.0 % of the original code. At an average there are 47.0 % reduction at an intermediate level, 6.5 % at assembly and 3.2 % at a link-time level. Relatively minor ones are dhry and fib. They use many 16-bit variables and exceed the 5-bit displacement used in index register addressing mode. In experiments, the general applications are evaluated. If evaluations are done on embedded application, the results will be better, because our techniques are targeted for embedded application such as global variable optimization and integer promotion elimination.

Table 2. Experimental results NO_OPT means no optimizations. IO, AO, LO mean intermediate optimization, assembly optimization, and link-time optimization respectively. For example, the last column shows the results of applying IO, AO, and LO. Each column shows the code size after optimization, not the reduction size. Bubble is a bubble sort program. Fib is a fibonacci permutation. Fact is a factorial calculation. Insert is a insertion sort program. Matmul is a matrix multiplication. Stan is a part of stanford benchmarks. Dhry is a part of dhrystone benchmarks.

Program	NO_OPT (Bytes)	IO (Bytes, %)	IO + AO (Bytes, %)	IO+AO+LO (Bytes, %)
Bubble	390	202 (51.8)	178 (45.6)	160 (41.0)
Fib	232	144 (62.1)	128 (55.1)	112 (48.3)
Fact	270	170 (62.9)	140 (51.8)	124 (45.9)
Insert	380	176 (46.4)	150 (39.5)	140 (36.8)
Matmul	1298	728 (56.0)	672 (51.7)	612 (47.1)
Stan	31578	16414 (51.9)	14502 (45.9)	13638 (43.1)
Dhry	1730	1204 (69.5)	1056 (61.0)	974 (56.0)
Total	35878	19038 (53.0)	16700 (46.5)	15560 (43.3)

In terms of cycle count, CalmRISC8 has a 3-stage pipeline. The first stage is instruction fetch stage, the second instruction decode and data memory access stage, and the final stage is execute and write-back stage. The number of clocks per instruction (CPI) is 1 except for long branches, which take 2 clock cycles. So reducing code size reduces cycle count also. Almost all optimizations contribute code size reduction. Procedural abstraction and library functions reduce code size, but increase execution cycle time by the generated function and call.

8. Conclusion

Embedded processors make compiler optimizations difficult. The extreme one is CalmRISC8. For embedded architectures, new optimization techniques should be applied. At intermediate level, assembly level, and link-time level the optimizations are performed, and passing information to later levels helps more optimizations being processed. Embedded architecture constraints can be overcome to great level. Several ideas can be extended to other embedded processors including DSP processors that have irregular structures like banked structure, and limited instruction set.

References

1. Samsung Corp.: CalmRISC8 Specification. Samsung Corp. (1999)
2. A. V. Aho, R. S. Sethi, and J. D. Ullman: Compilers: Principles, Techniques and Tools. Addison-Wesley (1985)
3. C. Fraser and D. Hanson: A Retargetable C Compiler: Design and Implementation. the Benjamin/Cummings, Redwood City, CA (1995)

4. G. J. Chaitin, M. A. Auslander, A. K. Chandra, J. Cocke, M. E. Hopkins, and P. Markstein: Register allocation via coloring. *Journal of Computer Languages*, 6 (1981) 45-57
5. P. Briggs, K. D. Cooper, K. Kennedy, and L. Torczon: Coloring Heuristics for register allocation. In *Proceedings of the ACM SIGPLAN '89 Conference on Programming Language Design and Implementation*, SIGPLAN Notices, 24(7) (1989) 275-284
6. F. Chow and J. Hennessy: Register allocation by priority-based coloring. *SIGPLAN Notices*, 19 (1984) 222-232
7. W. Hsu, C. N. Fischer, and J. R. Goodman: On the minimization of loads/stores in local register allocation. *IEEE Trans. on Software Engineering*, 15 (1989) 1252-1260
8. M. Farach and V. Liberatore: On Local Register Allocation. DIMACS Technical Report, July (1997) 93-33,
9. M. Z. Zastre: Compacting Object Code via Parameterized Procedural Abstraction. B.Sc. Thesis, Simon Fraser University (1993)
10. E. M. McCreight: A Space-Economical Suffix Tree Construction Algorithm. *Journal of the ACM*, Vol. 23, No. 2, April (1976) 262-272
11. A. Srivastava, and D. W. Wall: Link-Time Optimization of Address Calculation on a 64-bit Architecture. In *SIGPLAN Conference on Programming Language Design and Implementation*, (1993) 49-60
12. D W. Wall: Link-Time Code Modification. WRL Research Report 89/17, September (1989)
13. D. W. Wall: Experience with a Software-Defined Machine Architecture. *ACM Transactions on Programming Languages and Systems*, 14(3) (1992) 299-338