논문 2004-41SD-10-12

내장형 소프트웨어의 성능 향상을 위한 새로운 레지스터 할당 기법

(A New Register Allocation Technique for Performance Enhancement of Embedded Software)

이 종 열*

(Jong-Yeol Lee)

요 약

본 논문에서는 메모리 접근 연산을 레지스터 접근 연산으로 변환함으로써 레지스터를 할당하여 내장형 소프트웨어의 성능 향상을 도모할 수 있는 위한 레지스터 할당 기법을 제안한다. 제안된 방법에서는 프로파일링(profiling)을 통하여 메모리 트레이스(trace)를 얻는다. 그리고 각 함수의 수행 횟수에 대한 프로파일링 결과로부터 높은 동적 호출 횟수를 가지는 대상 함수를 선정하여 제안된 레지스터 할당 기법을 적용한다. 이와 같이 최적화의 대상이 되는 함수의 수를 줄임으로써 전체적인 컴파일 시간을 줄일 수 있다. 최적화대상 함수의 메모리 트레이스를 탐색하여 레지스터 접근 연산으로 변경될 경우 수행 사이클을 줄일 수 있는 메모리 접근 연산을 찾는다. 찾아진 메모리 접근 연산에 대해서는 컴파일러의 중간단계 코드를 수정하여 프로모션 레지스터(promotion register)를 할당한다. 이와 같은 과정을 거쳐 메모리 접근 연산이 프로모션 레지스터에 대한 접근 연산으로 대체되고 이로부터 성능향상을 얻을 수 있다. 제안된 레지스터 프로모션 기법을 ARM과 MCORE 프로세서용 컴파일러에 적용한 후 MediaBench와 DSPStone 벤치마크를 이용하여 cycle count를 비교함으로써 성능을 측정하였다. 그 결과 ARM과 MCORE에 대하여 평균 14%와 18%의 성능향상을 얻을 수 있었다.

Abstract

In this paper, a register allocation technique that translates memory accesses to register accesses is presented to enhance embedded software performance. In the proposed method, a source code is profiled to generate a memory trace. From the profiling results, target functions with high dynamic call counts are selected, and the proposed register allocation technique is applied only to the target functions to save the compilation time. The memory trace of the target functions is searched for the memory accesses that result in cycle count reduction when replaced by register accesses, and they are translated to register accesses by modifying the intermediate code and allocating promotion registers. The experiments where the performance is measured in terms of the cycle count on MediaBench and DSPstone benchmark programs show that the proposed method increases the performance by 14% and 18% on the average for ARM and MCORE, respectively.

Keywords : Embedded software, memory access, register access, profiling, register allocation

I. Introduction

In embedded systems based on programmable processors, high-level language compilers play an important role in the system design process. While assembly-level programming is still important to achieve optimized codes, high-level programming gains more and more acceptance for embedded processors to permit shorter design cycles, higher productivity and dependability, and higher opportu -nities for reuse. However, the code generated by compilers usually implies an overhead in code size and performance as compared to the hand-optimized assembly code. Although this overhead is acceptable in general-purpose computing, it is often not allowed for embedded systems.

* 정회원, 전북대학교 전자정보공학부
(Division of Electronics and Information, Chonbuk National University)
접수일자: 2004년4월20일, 수정완료일: 2004년9월22일

In order to minimize the overhead, embedded compilers have to pay higher attention to code optimization for both small code size and short execution time. As a consequence, a number of code optimization techniques have been developed for embedded processors. Most of these are low-level optimizations that exploit the detailed knowledge of the processor architecture to optimize machine code. For example, many low-level techniques were deve -loped for code selection, register allocation, and scheduling[1-3], memory access optimization[4], and optimization of address computations[5][6].

Since register allocation is one of the most impor -tant functions required in an optimizing compiler, many previous works[7] have dealt with this impor -tant problem. Most of the works mainly focused on the register allocation for scalar variables. Further -more, they treat non-scalar variables in a particularly naive fashion, making it impossible to determine when a specific element might be reused. Due to the incapability, conventional compilers alloca -te memory locations for global variables, structure, array and union variables. This is true even with highest optimizations. For example, when compiling a source code using GNU C Compiler (GCC)[8], such variables are allocated to memory even with "-O3" that is the highest optimization option. The structure and array variables are allocated to memory since they are treated as a whole. Compiler optimizations do not usually treat the fields of structures and the elements of arrays as separate variables. In case of global variables, they are allocated to memory because they can be accessed in two or more func -tions and registers cannot be allocated beyond function boundary. For array variables, the previous works have focused on the dependency analysis of array variables used in loops. A representation method of array access patterns and a dependency analysis were presented without considering a regis -ter allocation.[9] In other works[10][11], pointer analysis was exploited to treat pointer-valued variables and arrays in C language. As the pointer analysis used in the works was simple, their results were not as good as we expected. In this paper we present a new register allocation technique to achieve higher perfor -mance of embedded software. The proposed register allocation is to translate frequently accessed memory locations to register accesses, and can be regarded as an optimization because instructions generated to access data objects in registers are more efficient than the case that the objects are in memory. We need to determine which variables can be safely kept in registers and to rewrite the code to keep the found variables in registers. Based on the results of profiling, the proposed register allocation technique identifies code sections in which it is safe to place the value of a data object in a register, and then promotes memory locations that are frequently accessed in the code segments to registers. As the proposed technique is independent of the type of variables, it can be applied to non-scalar variables as well as scalar ones.

Since the proposed register allocation can be applied after conventional optimization developed in the previous works, it can be efficiently used for the optimization of embedded software as a complemen -tary tool to the previous works. Furthermore, since the proposed method uses the profiling, it can register-promote pointers effectively. In the proposed method the trace of memory accesses are generated in the profiling and analyzed to perform the register allocation.

The rest of this paper is organized as follows. In Section II, a motivating example is given and the proposed register allocation is described. After showing the experimental results in Section III, conclusions are addressed in Section IV.

II. The Proposed Register Allocation

An example of register allocation is shown in Fig. 1, where memory is allocated for array variables, A and B. In Fig. 1(b), registers, R0-R3, are reserved for function arguments and are not allocated in register allocation. Registers, R4-R13, are used in register allocation and the remaining two registers, R14 and

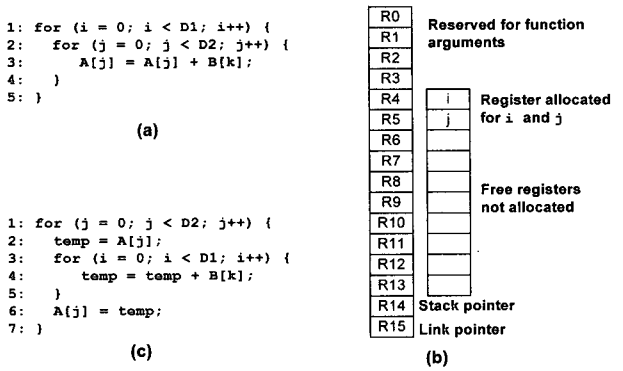

그림 1. 레지스터 할당 예제 (a) 소스 코드. (b) 레지스터 할당 결과. (c) A[j]에 레지스터를 할당하기 위한 변환 결과의 소스 코드 표현

Fig. 1. Example of register allocation. (a) C source code. (b) Register allocation result. (c) Source code representation of a transformation result for the register allocation to A[i]s.

R15, are used as stack pointer and link register, respectively. We can see that two registers, R4 and R5, are allocated for local variables, i and j. R6-R13 are free registers that are not allocated.

Although there are free registers, memory locations are allocated for the arrays as a result of the inability of compilers stated as above. In Fig. 1(a), we can see that the element of A accessed multiple times in the inner loop is independent of the index of the inner loop. Therefore, by moving the value of the element into a scalar variable before the inner loop and restoring the variable back to the memory location after the inner loop, we can replace the memory access in the inner loop by a register access. The source code representation of the result after a transformation for the register allocation to array elements is shown in Fig. 1(c), where a new temporary scalar variable that will be allocated to a register is added.

The cycle counts, TC_{before}, before the register allocation can be estimated as follows with assuming the cycle counts of memory accesses, C_r and C_w, are two and that of addition, C_{add}, is one.

$$B_{before} = 2 \times C_r + 1 \times C_{add} + 1 \times C_w = 7 \quad (1)$$

$$TC_{before} = B_{before} \times D1 \times D2 = 7 \times D1 \times D2 \quad (2)$$

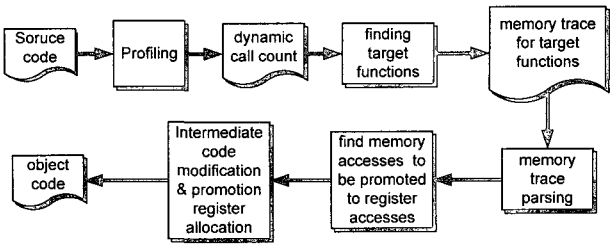

그림 2. 제안된 레지스터 할당 방법

Fig. 2. Overview of the proposed register allocation method.

where B_{before} is the estimated cycle count of the loop body.

Since the memory access, the access to A[j], is replaced by a register access after the register allocation, the access to temp, the total cycle count, TC_{after}, is calculated as follows.

$$B_{after} = 1 \times C_r + 1 \times C_{add} = 3 \quad (3)$$

$$TC_{after} = D1 \times [1 \times C_r + D2 \times B_{after} + 1 \times C_w] = 4 \times D1 + 3 \times D1 \times D2 \quad (4)$$

$$TC_{before} - TC_{after} = 4 \times D1 \times (D2 - 1) \quad (5)$$

By calculating the difference of two total cycle counts, TC_{before} and TC_{after}, as shown in equation (5), we can determine whether the register allocation is effective or not. It should be noted that we describe for clear understanding as if the source code is modified directly. In the real implementation, the modification is applied to the intermediate code of compilers such as the RTL of GCC. Therefore, the source code is never changed but the intermediate code generated after parsing the source code is modified to account for the proposed register allocation to be accommodated.

The technique proposed in this paper performs register allocation for non-scalar variables in a systematic way. Fig. 2 shows the overall flow of the proposed method. In the proposed method, the infor-mation needed for the optimizations is obtained by profiling the source code. The advantage of profiling is that it can produce more accurate result than the static analysis such as pointer analysis. For register allocation, memory trace is generated and analyzed to find memory locations accessed frequently. They can

be replaced with register accesses to reduce cycle count. After finding such memory locations, the intermediate code is modified to move the data in the memory locations to promotion registers reserved to be allocated for the found memory locations. The promotion registers are reserved only in the functions where the dynamic call count is high and hence, the proposed method is applied. Therefore, in the functions where the proposed method is not exploited, all the machine registers are used in the register allocation step. This is because the register allocation is performed as an intra-procedural optimization. Note that, in the proposed method, pointers can be optimized without using pointer analysis because memory trace is used.

Unlike the conventional register promotion, the proposed method uses two techniques to enhance performance by allocating registers to the frequently accessed memory locations. In contrast to the conventional register promotion where the performance enhancement is estimated in a static analysis to find the target memory locations of register promotion, the proposed method uses the profiling technique to estimate the performance enhancement more accurately. By allocating some of machine registers for the register promotion beforehand, the frequently accessed memory locations can be allocated to the reserved registers irrespective of the result of the general register allocation.

1. Source Code Profiling

As the first step of the proposed register allocation technique, the source code is profiled to find target functions to which the proposed technique is applied. To maximize the effect of the register allocation and reduce the compilation time, the only functions with a high dynamic call count are considered.

For the profiling, instruction-set simulators are modified to report dynamic calls and generate me -mory trace. The address and program counter (PC) values are dumped for each load or store instruction with a field indicating the type of memory operation (load or store).

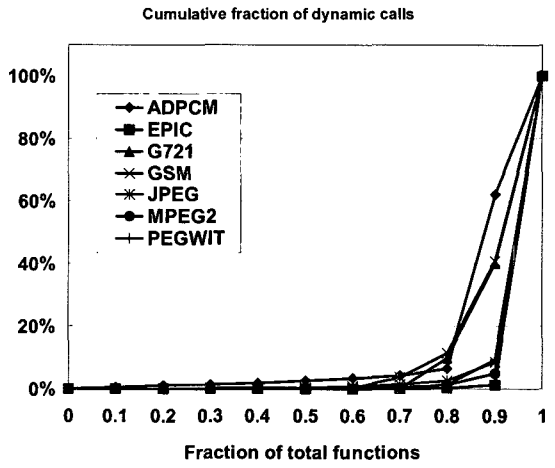

그림 3. 동적 함수 호출의 분포
Fig. 3. Distribution of dynamic function call.

An advantage of the profiling is that the number of functions to be considered in the optimization routine is reduced by selecting only the functions called frequently. As can be seen in Fig. 3, most of dynamic function calls are dedicated to only a small portion of the whole functions. As a consequence, significant improvement can be achieved with applying optimizations to the heavily-called functions. In our experience, only one or two functions take 95% of total dynamic calls in all the benchmarks used in the experiments. Another advantage is that better results than static pointer analysis can be obtained in the presence of pointers because the aliasing problem can be eliminated by analyzing addresses in the memory trace.

2. Iteration Tree Construction

The memory trace of heavily executed functions is parsed to find whether the proposed register alloca -tion can increase performance. The first step of the implemented memory trace parser reads the trace into memory and the following steps are performed on the trace in memory because file I/O is the most time -consuming task.

In memory trace parsing, for each memory access in the intermediate code, a corresponding memory address is found. The type of a memory access is also determined. This is achieved by inserting

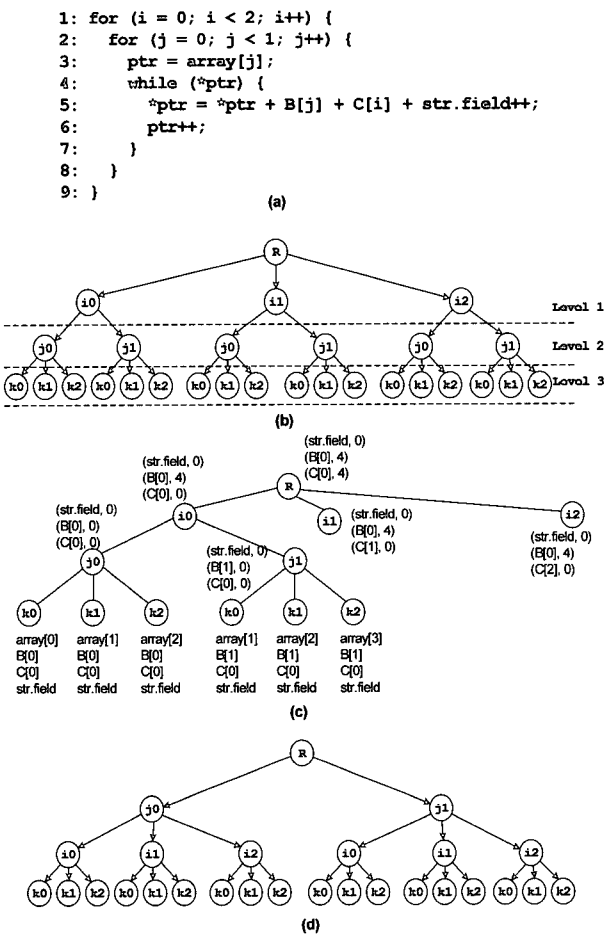

그림 4. 반복 트리 (a) 소스 코드 예 (b) 반복 트리 (c) CAS 계산 (d) 루프 교환 후의 반복 트리

Fig. 4. Iteration tree. (a) An example source code. (b) An iteration tree. (c) CAS calculation. (d) An iteration tree after loop interchanging.

debugging information into the executable when com-piling the source code. A source-level debugger uses the debugging information to find source lines corresponding to a given PC value. The debugging information is also used to find the type of a memory access in a source-level debugger in order to show the value of a user-defined variable in an appropriate form. In the proposed method the debugging informa-tion is used to map a PC value to a memory operation in the intermediate code and find the type of a memory access.

In addition, loops can be found by examining PC values in trace and debugging information. For the found loops, an iteration tree is constructed where nodes represent instances of loop iterations. Each node has a pointer to a loop and the index value of the corresponding iteration. Fig. 4 shows an iteration tree example. In Fig. 4(b), the nodes, i0, i1, and i2, represent the iterations of the outer-most loop whose index variable is i. For a loop without an index variable a pseudo index variable is inserted. For example, the inner-most loop in Fig. 4 is a while loop that has no index variable. The iterations of the while loop with a pseudo index variable, k, are shown at the leaf nodes of Fig. 4(b).

3. Code Generation

The next step is to find the memory accesses to be promoted to register accesses by examining the iteration tree. The candidate memory accesses for a promotion to registers are the accesses whose addresses are constant over all the iterations of a loop. We assume that each memory address is asso-ciated with a PC value. To find candidate memory accesses, the nodes are levelized by their distance to the root node. For the iteration tree of Fig. 4(b), the first-level nodes are i0, i1 and i2. A *canonical address set (CAS)* is calculated for a node. If memory addresses with the same PC value have a uniform stride, the addresses are called canonical in this paper and can be represented as a form of *(starting address, stride)*. The CAS of a leaf node contains all of the memory addresses accessed in the iteration corresponding to the leaf node, as there is one memory access for a PC value. As in this case, it is difficult to define the stride, but we assume every memory address has a uniform stride initially. Therefore, a CAS contains only canonical addresses, which means a set of memory addresses that have the same PC but do not have a uniform stride is not considered in the proposed register allocation.

A CAS of a non-leaf node, CAS_n, with D immediate descendants is calculated by using the following equation.

$$CAS_n = CAS_{n,initial} \odot \{ \overset{D-1}{\underset{i=0}{\odot}} CAS_i \} \quad (6)$$

where CAS_i and $CAS_{n,initial}$ are the CAS of an imme-diate descendant of the non-leaf node and the initial

value of the CAS of the non-leaf node, respectively. The operator ⊙ finds the canonical address when applied to the elements of the immediate descendant CASs. For the CAS elements with the same PC value, the ⊙ operator is calculated by using the following equations. For any starting addresses, A_1 and A_2, and strides, S_1 and S_2,

$$(A_1, S_1) \odot (A_1, S_1) = (A_1, S_1) \quad (7)$$

$$(A_1, 0) \odot (A_1 + S_1, 0) = (A_1, S_1) \quad (8)$$

$$(A_1, S_1) \odot (A_1 + m \times S_1, 0) = (A_1, S_1) \quad (9)$$

$$(A_1, S_1) \odot (A_1 + m \times S_1, S_1) = (A_1, S_1) \quad (10)$$

$$(A_1, S_1) \odot (A_2, S_1) = (A_1, S_1) \odot (A_2, S_2) = \varnothing \quad (11)$$

$$(A_1, S_1) \odot (A_1, S_2) = \varnothing \quad (12)$$

where m is an integer. Equations (11) and (12) hold because the CAS elements represent non-uniform strides. An example of CAS calculation is shown in Fig. 4(c), where each leaf node has memory accesses in the corresponding iteration and non-leaf nodes have their CASs. The CAS of i0 contains **(B[0], 4)** that is calculated from **(B[0], 0)** in the CAS of j0 and **(B[1], 0)** in the CAS of j1 by equation (8). Since in the descendants of i0, **B[0]** and **B[1]** are accessed in sequence, the stride is four, which is the size of elements in B.

How to find candidate memory accesses is explained using an example. Let A1, A2, and A3 be canonical addresses in the CASs of i0, i1 and i2, respectively, in Fig. 4(b). When the following equation holds

$$A2 - A1 = A3 - A2 = d \quad (13)$$

and the number of traces is equal to the statically determined number of iterations, the addresses in iterations can be represented as an arithmetic series

$$A1 = A1 + d*0 \quad (14)$$

$$A2 = A1 + d*1 \quad (15)$$

$$A3 = A1 + d*2 \quad (16)$$

In general, if all the differences of two consecutive addresses are identical, the addresses form an arithmetic series and the address in i-th iteration can be represented as a general term,

$$A(i) = A1 + d*i \quad (17)$$

where A1 is the address in the first iteration and d is a stride.

If the stride of memory addresses associated with the same PC value is zero in equation (17), that is, memory addresses are constant, the corresponding me-mory accesses might be replaced by register accesses with additional load and store instructions. The additional instructions are placed outside the loop corresponding to the level on which the general term is calculated. If equation (17) holds for the first-level CASs in Fig. 4(b) with a zero stride, a load instruction and a store instruction from/to the canonical address are placed before and after the outer-most loop, respectively. When d is not zero, the address changes with a equal stride, d. In this case, the address can be efficiently implemented by using auto-increment or auto-decrement addressing mode.

In some cases more cycle count reduction can be achieved by interchanging loops. When two or more nodes have the same name in a level and the CASs of the nodes have common canonical addresses, by interchanging the loops corresponding to the level and its upper level the constant addresses can be exploited in the proposed register allocation resulting in cycle count reduction. If CASs of j0s in Fig. 4(b) have the same constant address, A1, and j1s also have the same constant address, A2, by inter-changing the first and second loops corresponding to Level 1 and Level 2, respectively, the canonical addresses will be found at Level 1. As canonical addresses are found in the upper level after the loops are interchanged, the additional load and store instructions are inserted at outer position, and hence, the amount of cycle count increased by the instruc-tions can be reduced in the modified code. The iteration tree with the loop interchanging is shown in Fig. 4(d).

After finding candidate memory accesses for the

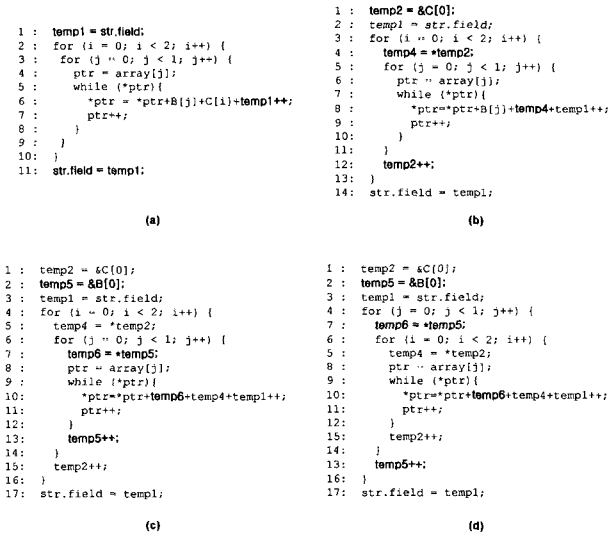

그림 5. 코드 변환 (a) 구조체 변수 str.field의 주소는 모든 루프 반복에서 상수임. (b) C[i]의 주소는 인덱스 변수 i로 표시할 수 있는 등차수열임. (c) 루프 교환을 수행하지 않은 코드 변환 결과 (d) 루프 교환을 수행한 후의 코드 변환 결과

Fig. 5. Code modification. For clear understanding the modification results are shown using source codes instead of the intermediate code on which the modification is performed. (a) The address of a structure variable, str.field, is constant in the iterations of all loops. (b) The addresses of C[i] form an arithmetic series with index variable i. (c) Code modification without interchanging the first two for loops. (d) Code modification after interchanging the first two for loops.

promotion to registers, we can calculate the cycle count difference between the original code and the register promoted code by taking into account the execution counts of the loops and functions. If the difference is positive, the memory accesses are replaced by register accesses.

The first step of the replacement is to modify the intermediate code. In this step, additional variables are defined and codes that move values from/to the additional variables are inserted. Fig. 5 shows the source modification results of the example code in Fig. 4(a). At each level of the iteration tree, we search for the same canonical addresses. In Level 1 of Fig. 4(b), the memory access to str.field is a canonical address. Therefore, a load instruction to a temporary variable, temp1, and a store instruction from the temporary variable are inserted before and after of the outer-most for loop, respectively. Another temporary variable, temp3, is used to promote the accesses to B[j] in Fig. 5(c) because the addresses of B[j] are constant over the iterations of the inner-most while loop.

The addresses of C[i] in Fig. 4(a) form an arithmetic series with the index variable i. The first address is the address of the first element, C[0]. The difference between the first and second addresses is C[1] - C[0], which is four if C is an array of four-byte integers. Therefore, the address increases by four at each iteration. The code is modified by inserting a statement that loads the address to a temporary variable, temp2. At the end of the first for loop, the address is incremented by four. In the new statement, temp2++ is used instead of temp2 = temp2 + 4 because temp2++ in C language statement is interpreted as the increment of the address by the size of the array elements. The resulting code is shown in Fig. 5(b).

We can calculate the cycle count before and after loop interchanging shown in Fig. 5(c) and (d) as $C_{before} = 3x2x(M+WxR)$ and $C_{after} = 2x(M+3xWxR)$ where M is the cycle count of a memory access, R is that of a register access, and W is the number of iterations of the inner-most while loop. If we assume that the values of M, R, and W are five, three, and one, respectively, C_{before} = 48 and C_{after} = 28. In this calculation, the reduction in cycle count is about 42%.

The promotion registers that are special registers dedicated for the promotion of memory accesses to register accesses are allocated for the compiler -inserted variables, temp1 and temp3. In most cases, the promotion registers need not be saved and restored to preserve their values, since the promotion registers are not used in general register allocations. If there are compiler-inserted variables not mapped to promotion registers because of the restricted number of the promotion registers, the compiler may allocate free registers for the variables.

Since the case that all the registers are used in register allocation is rare in general compilers, some of the registers can be used as promotion registers. If some registers are reserved as promotion registers,

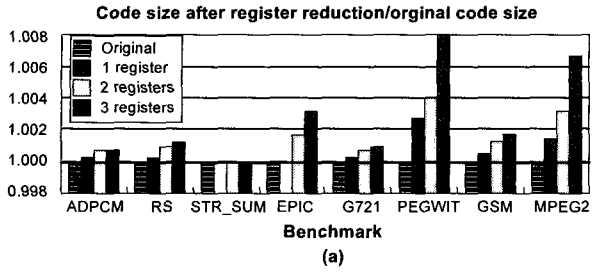

그림 6. ARM의 레지스터 축소 효과 (a) 레지스터의 수 감소에 따른 크기 증가 (b) 레지스터의 수의 감소에 따른 cycle count 증가

Fig. 6. Effects of register reduction in ARM. (a) Code size increase when the number of available registers are reduced by one, two and three. (b) Cycle count increase when the number of availa -ble registers are reduced by one, two and three.

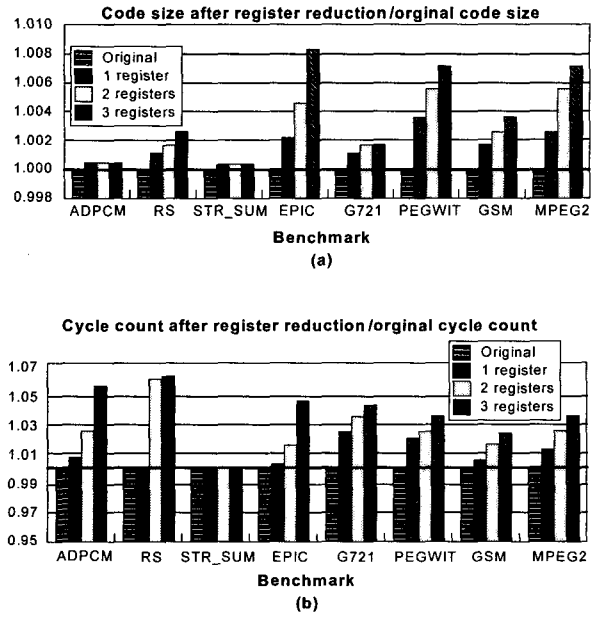

그림 7. MCORE의 레지스터 축소 효과 (a) 레지스터의 수 감소에 따른 크기 증가 (b) 레지스터의 수의 감소에 따른 cycle count 증가

Fig. 7. Effects of register reduction in ARM. (a) Code size increase when the number of available registers are reduced by one, two and three. (b) Cycle count increase when the number of available registers are reduced by one, two and three.

the maximum number of registers that can be involved in general register allocation is reduced, which may lead to significant effects on the code size and performance. Therefore, the number of promotion registers must be determined by investigating such effects.

Ⅲ. Experimental Results

We implemented the proposed register allocation method using GCC and performed experiments for ARM and MCORE. ARM and MCORE are selected because they are typical embedded processors. To measure the cycle count variation, ARMulator and MCORE instruction-set simulator (ISS) are used. As listed in Table 1, benchmark programs used in the experiments are MediaBench[12] and DSPstone[13]. Some of the programs, DOT_PRODUCT, FIR2DIM, MATRIX_MUL and STR_SUM use pointers heavily and are included to show that the proposed method can handle pointers appropriately.

표 1. 사용된 벤치마크 프로그램

Table 1. Summary of Benchmark Programs.

Benchmark	Description
RS	Reed-Solomon coder/decoder
STR_SUM	Summation of structures
PEGWIT	Public key encryption
ADPCM	ADPCM coder/decoder
MPEG2	MPEG2 encoder
DOT_PRODUCT	Dot product
FIR2DIM	Two-dimensional FIR filter
MATRIX_MUL	Matrix multiplication

표 2. 제안된 레지스터 할당이 적용된 함수의 수

Table 2. Number of functions to which the proposed register allocation is applied.

Benchmark	Number of functions
RS	2
STR_SUM	1
PEGWIT	2
ADPCM	2
MPEG2	1
DOT_PRODUCT	1
FIR2DIM	1
MATRIX_MUL	1

First, we performed experiments to determine the number of promotion registers. We generated execu -tables by restricting the number of registers. Fig. 6 and Fig. 7 show the effects of register reduction on the code size and cycle count for ARM and MCORE, which are obtained with a modified GCC. Based on the results in Fig. 6 and Fig. 7, we can determine the number of promotion registers. The amount of code size increase is less than 1% for both processors until the number of registers available for register allocation is reduced by three. However, the register reduction affects cycle count more significantly. In ARM, the cycle count increases more than 30% when the number of registers is reduced by three. In Fig. 6, we can see that reserving two registers as promotion registers in ARM does not impose great penalty on code size and cycle count. For MCORE, there is no steep increase in cycle count for three promotion registers. Therefore, two and three promotion registers are assigned for ARM and MCORE, respectively.

We compiled the benchmark programs using the implemented compiler for ARM and MCORE with the determined number of promotion registers. In the experiments, only one or two functions are selected and the proposed register allocation is applied as summarized in Table 2. By executing the generated executables on ARMulator and MCORE ISS, the cycle counts are measured. Fig. 8 shows the code size and cycle count variation after the proposed register allocation. The maximum code size increase is about 1% for MCORE, and the code size remains almost constant in case of ARM. However, the cycle count is reduced for both ARM and MCORE. The average cycle count reductions are about 14% and 18% for ARM and MCORE, respectively. The maximum cycle count reductions are 30% and 35% for ARM and MCORE, respectively. Fig. 8 also indicates that the proposed method can handle pointers effectively because the average cycle count reductions for programs that heavily use pointers are about 15.2% and 20.5% for ARM and MCORE, respectively.

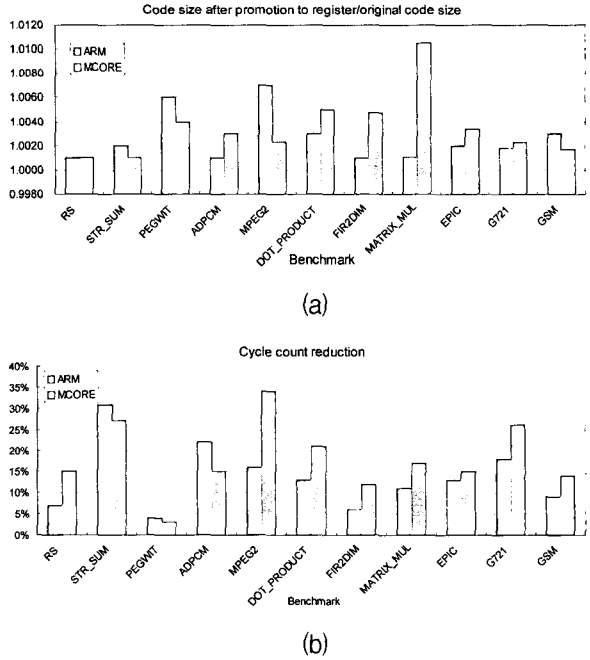

그림 8. 코드 크기 변화와 cycle count 감소 (a) 코드 크기 변화 (b) cycle count 감소

Fig. 8. Code size variation and cycle count reduction after the proposed register allocation. (a) Code size variation. (b) Cycle count reduction.

IV. Conclusions

In this paper, we have presented a register alloca -tion technique to enhance performance by promoting memory accesses to register accesses. In the pro -posed method, a given source code is profiled to generate a memory trace. The profiling result is used to find target functions with high dynamic call counts, and the proposed register allocation is applied only to the target functions to save the compilation time. By examining the memory trace of the target functions, we search for memory accesses that can result in cycle count reduction if changed to register accesses. Such a memory access is replaced by a register access by modifying the code and allocating a promotion register. The experimental results show that the proposed method is effective in that average cycle count reduction is about 14% with less than 1% increment of the code size after the proposed register allocation is applied.

참 고 문 헌

[1] C. Liem, T. May, and P. Paulin, "Instruction-Set Matching and Selection for DSP and ASIP Code Generation," In European Design and Test Conference ED & TC), pp. 31-37, 1994.

[2] G. Araujo and S. Malik, "Optimal Code Genera -tion for Embedded Memory Non-Homogeneous Register Architecture," In 8th Int. Symp. On System Synthesis, pp. 13-15, 1995.

[3] S. Liao, S. Devadas, K. Keutzer, and S. Tjiang, G. Araujo and S. Malik, "Instruction Selection Using Binate Covering for Code Size Optimiza -tion," In Int. Conf. on Computer-Aided Design (ICCAD), pp. 5-9, 1995.

[4] A. Sudarsanam and S. Malik, "Memory Bank and Register Allocation in Software Synthesis for ASIPs," In Int. Conf. on Computer-Aided Design (ICCAD), pp. 393-399, 1995.

[5] S. Liao, S. Devadas, K. Keutzer, S. Tjiang, and A. Wang, ''Storage Assignment to decrease code size'', *ACM Trans. Program. Lang. and Sys.* vol. 18, no. 3, pp. 186-195, May 1996.

[6] R. Leupers and P. Marwedel, "Algorithms for Address Assignment in DSP Code Generation," In Int. Conf. on Computer-Aided Design (ICCAD), pp. 109-112, 1996.

[7] A. Aho, R. Sethi, and J. Ullman, *Compilers - Principles, Techniques, and Tools*. Reading, MA: Addison-Wesley, 1986.

[8] R. M. Stallman. *Using and Porting GNU CC.* [Online]. Available: http://gcc.gnu.org/onlinedocs/gcc-2.95.3/gcc.html.

[9] M. Wolfe, *High Performance Compilers for Parallel Computing*. Redwood City, CA: Addison -Wesley, 1996.

[10] M. hind, "Pointer Analysis: Haven't We Solved This Problem Yet?" In PASTE, pp. 54-61, 2001.

[11] R. P. Wilson and M. S. Lam, "Efficient Contex -Sensitive Pointer Analysis for C Programs," In SIGPLAN PLDI, pp. 1-12, 1995.

[12] Chunho Lee, M. Potkonjak, W. H. Mangione -Smith, MediaBench. [Online]. Available: http://www.cs.ucla.edu/ leec/mediabench/

[13] V. Zivojnovici, J. Martinez Velarde, and C. Schlager, "DSPstone : A DSP-oriented Bench -marking Methodology," In Proc. Int. Conf. On Signal Processing and Technology, Dallas, pp. 715-720, 1994.

저 자 소 개

이 종 열(정회원)

1993년 한국과학기술원 전자전산학과 졸업 (B.S.).
1996년 한국과학기술원 전자전산학과 졸업 (M.S.).
2002년 한국과학기술원 전자전산학과 박사 (Ph.D.).
2002년 8월~2003년 9월 하이닉스 반도체 선임연구원
2003년 10월~2004년 2월 한국과학기술원 BK21 초빙교수
2004년 3월~현재 전북대학교 전자정보공학부 전임강사

<주관심분야: SoC 설계, 내장형 프로세서 설계, 내장형 소프트웨어>