Available online at www.sciencedirect.com
SCIENCE DIRECT
Microprocessors and Microsystems 29 (2005) 177–187
MICROPROCESSORS AND MICROSYSTEMS
www.elsevier.com/locate/micpro

Performance enhancement of embedded software based on new register allocation technique ☆

Jong-Yeol Lee[a,*], Seong-IK Cho[a], In-Cheol Park[b]

[a]Division of Electronics and Information Engineering, Chonbuk National University, 664-14 1ga Duckjin-Dong, Duckjin-Gu, Jeonju, Jeonbuk 561-756, South Korea
[b]Department of Electrical Engineering and Computer Science, Korea Advanced Institute of Science and Technology, 373-1 Guseong-dong, Yuseong-gu, Daejeon 305-701, South Korea

Received 22 April 2004; revised 31 August 2004; accepted 3 October 2004
Available online 29 October 2004

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 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

1. 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 opportunities 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. This overhead is not acceptable in embedded systems that usually have stringent restrictions on code size and performance because codes on embedded systems often must run with real-time constraints and limited hardware resources. A real-time performance requirement is one where a segment of the application has an absolute maximum execution time that is allowed. For example, in a digital set-top box the time to process each video frame is limited because the processor must accept and process the next frame shortly. Many embedded applications also impose restrictions on the size of code. For example, in a portable embedded system where the power consumption depends, in part, on the amount of memory the embedded system contains, the code should be optimized in size to save power consumption because code is often stored in read-only memory (ROM) and the size of code determines the size of the ROM.

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 developed for

☆ This paper was supported (in part) by the research funds of Chonbuk National University.
* Corresponding author. Tel.: +82 632 704 140; fax: +82 632 702 394.
E-mail address: jong@chobuk.ac.kr (J.-Y. Lee).

0141-9331/$ - see front matter © 2004 Elsevier B.V. All rights reserved.
doi:10.1016/j.micpro.2004.10.001

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 important functions required in an optimizing compiler, many previous works such as [7] have dealt with this important problem. Most of the works mainly focused on the register allocation for scalar variables. Furthermore, they treat non-scalar variables in a particularly naive fashion, making it impossible to determine when a specific element might be reused. Due to this incapability, conventional compilers allocate 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) [11], such variables are allocated to memory even with '-O3' which 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 functions and registers cannot be allocated beyond function boundaries. For array variables, the previous works have focused on the dependency analysis of array variables used in loops. In [8], a representation method of array access patterns and a dependency analysis were presented without considering a register allocation. In other works [9,10], 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. Complementary to previous register allocation techniques, in this paper, we present a new register allocation technique to achieve higher performance 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.

The rest of this paper is organized as follows. In Section 2, a motivating example is given and the proposed register allocation is described. After showing the experimental results in Section 3, conclusions are addressed in Section 4.

2. 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 `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

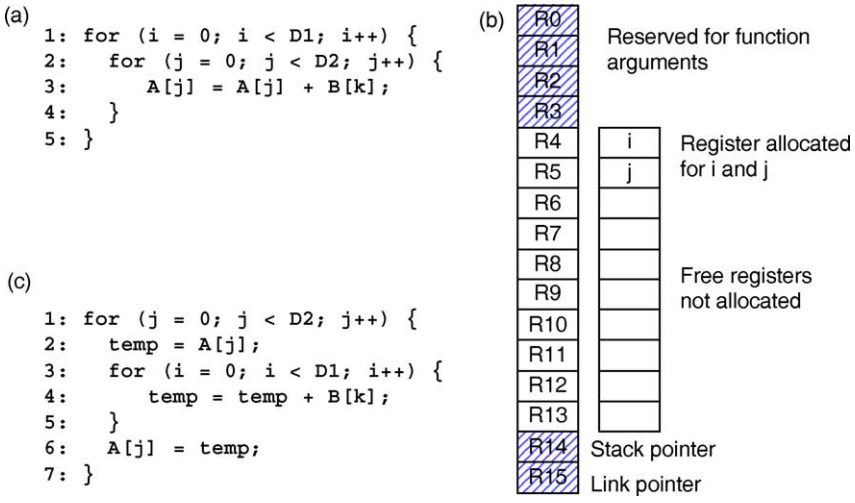

Fig. 1. Example of register allocation. (a) C source code. (b) Register allocation result. (c) Source code representation of a transformation result for the register allocation to `A[j]`s.

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

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

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

where C_r and C_w are the average cycle counts of memory accesses with cache and B_{before} is the estimated cycle count of the loop body. The average cycle counts can be calculated by averaging the cache miss cycles, $C_{r,miss}$ and $C_{w,miss}$, and the cache hit cycles, $C_{r,hit}$ and $C_{w,hit}$ as follows:

$$C_r = h \times C_{r,hit} + (1 - h) \times C_{r,miss} \quad (3)$$

$$C_w = h \times C_{w,hit} + (1 - h) \times C_{w,miss} \quad (4)$$

where h is a cache hit rate whose value is between 0 and 1. By assuming the average cycle counts of memory accesses, C_r and C_w, are two and that of addition, C_{add} is one, $B_{before} = 7$ and $TC_{before} = 7 \times D1 \times D2$.

Unlike general-purpose computer systems that invariably have caches, embedded systems usually have no data cache. This is because embedded systems are often placed in real-time environments where a set of tasks must be completed every time period and in such situations performance variability is of more concern than average-case performance. Since caches are used to improve average-case performance at the cost of greater variability, caches are not a must for real-time computing. When cache is not present, Eqs. (3) and (4) can be used with setting h to zero.

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:

$$B_{after} = 1 \times C_r + 1 \times C_{add} = 3 \quad (5)$$

$$TC_{after} = D1 \times [1 \times C_r + D2 \times B_{after} + 1 \times C_w] = 4 \times D1 + 3 \times D1 \times D2 \quad (6)$$

$$TC_{before} - TC_{after} = 4 \times D1 \times (D2 - 1) \quad (7)$$

By calculating the difference of two total cycle counts, TC_{before} and TC_{after}, as shown in Eq. (7), 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 information 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. Note that, in the proposed method, pointers can be optimized without using pointer analysis because memory trace is used.

2.1. 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 memory trace.

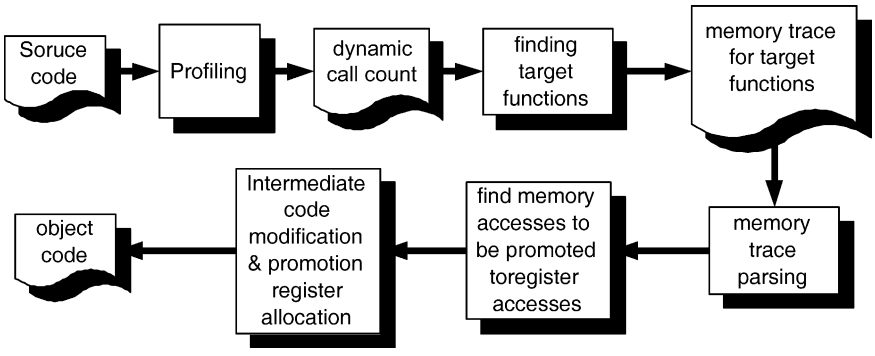

Fig. 2. Overview of the proposed register allocation method.

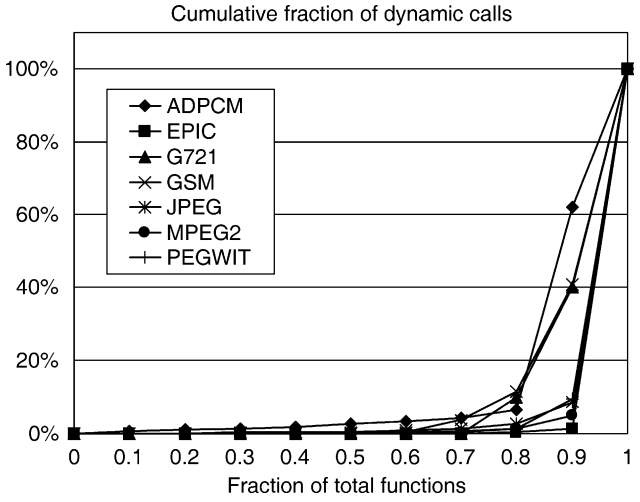

Fig. 3. Distribution of dynamic function calls.

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).

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.

2.2. Iteration tree construction

The memory trace of heavily executed functions is parsed to find whether the proposed register allocation 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. This is achieved by inserting debugging information into the executable when compiling the source code. A source-level debugger uses the debugging information to find source lines corresponding to a given PC value. In the proposed method the debugging information is used to map a PC value to a memory operation in the intermediate code.

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).

2.3. 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 associated 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 is calculated as $\mathrm{CAS}_n \odot \{\odot \mathrm{CAS}_i\}$, where $\odot$ is an operator that finds the canonical addresses, CAS_n is the CAS of the non-leaf node and CAS_i is the CAS of an immediate descendant of the non-leaf node. 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`. 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

$$\mathtt{A2} - \mathtt{A1} = \mathtt{A3} - \mathtt{A2} = d \tag{8}$$

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

$$\mathtt{A1} = \mathtt{A1} + d \times 0 \tag{9}$$

(a)

```
1: for (i = 0; i < 2; i++) {
2:   for (j = 0; j < 1; j++) {
3:     ptr = array[j];
4:     while (*ptr) {
5:       *ptr = *ptr + B[j] + C[i] + str.field++;
6:       ptr++;
7:     }
8:   }
9: }
```

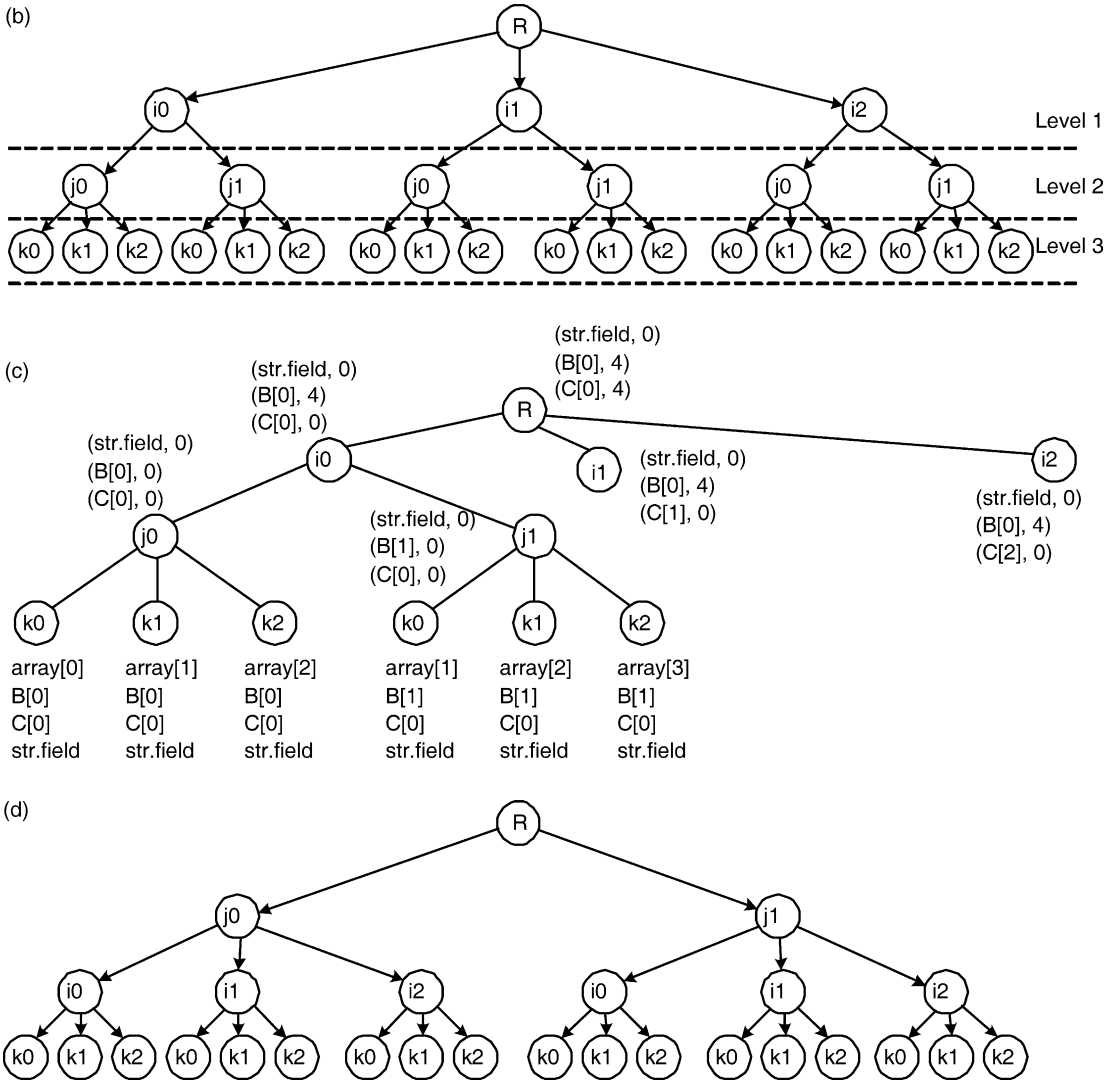

Fig. 4. Iteration tree. (a) An example source code. (b) An iteration tree. (c) CAS calculation. (d) An iteration tree after loop interchanging.

$$\mathrm{A2} = \mathrm{A1} + d \times 1 \tag{10}$$

$$\mathrm{A3} = \mathrm{A1} + d \times 2 \tag{11}$$

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

$$\mathrm{A}(i) = \mathrm{A1} + d \times i \tag{12}$$

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 Eq. (12), that is, memory addresses are constant, the corresponding memory 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 Eq. (12) 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,

(a)
```
1 :  temp1 = str.field;
2 :  for (i = 0; i < 2; i++) {
3 :   for (j = 0; j < 1; j++) {
4 :     ptr = array[j];
5 :     while (*ptr){
6 :       *ptr = *ptr+B[j]+C[i]+temp1++;
7 :       ptr++;
8 :     }
9 :   }
10:  }
11:  str.field = temp1;
```

(b)
```
1 :  temp2 = &C[0];
2 :  temp1 = str.field;
3 :  for (i = 0; i < 2; i++) {
4 :    temp4 = *temp2;
5 :    for (j = 0; j < 1; j++) {
6 :      ptr = array[j];
7 :      while (*ptr){
8 :        *ptr=*ptr+B[j]+temp4+temp1++;
9 :        ptr++;
10:      }
11:    }
12:    temp2++;
13:  }
14:  str.field = temp1;
```

(c)
```
1 :  temp2 = &C[0];
2 :  temp5 = &B[0];
3 :  temp1 = str.field;
4 :  for (i = 0; i < 2; i++) {
5 :    temp4 = *temp2;
6 :    for (j = 0; j < 1; j++) {
7 :      temp6 = *temp5;
8 :      ptr = array[j];
9 :      while (*ptr){
10:       *ptr=*ptr+temp6+temp4+temp1++;
11:       ptr++;
12:     }
13:     temp5++;
14:   }
15:   temp2++;
16:  }
17:  str.field = temp1;
```

(d)
```
1 :  temp2 = &C[0];
2 :  temp5 = &B[0];
3 :  temp1 = str.field;
4 :  for (j = 0; j < 1; j++) {
7 :    temp6 = *temp5;
6 :    for (i = 0; i < 2; i++) {
5 :      temp4 = *temp2;
8 :      ptr = array[j];
9 :      while (*ptr){
10:       *ptr=*ptr+temp6+temp4+temp1++;
11:       ptr++;
12:     }
15:     temp2++;
14:   }
13:   temp5++;
16:  }
17:  str.field = temp1;
```

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.

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 `j0`s in Fig. 4(b) have the same constant address, `A1`, and `j1`s also have the same constant address, `A2`, by interchanging 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 instructions 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 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

Table 1
Summary of benchmark programs

Benchmark	Description
RS	Reed-Solomon code/decoder
STR(SUM	Summation of structures
PEGWIT	Public key encryption
ADPCM	ADPCM coder/decoder
MPEG2	MPEG2 encoder
DOT_PDT	Dot product
FIR2DIM	Two-dimensional FIR filter
MAT_MUL	Matrix multiplication

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 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 follows

$$C_{\text{before}} = 3 \times 2 \times (M + W \times R) \tag{13}$$

$$C_{\text{after}} = 2 \times (M + 3 \times W \times R) \tag{14}$$

where M is the average cycle count of a memory access calculated using Eqs. (5) and (4) and, R is the cycle count 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:

$$C_{\text{before}} = 48 \tag{15}$$

$$C_{\text{after}} = 28 \tag{16}$$

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

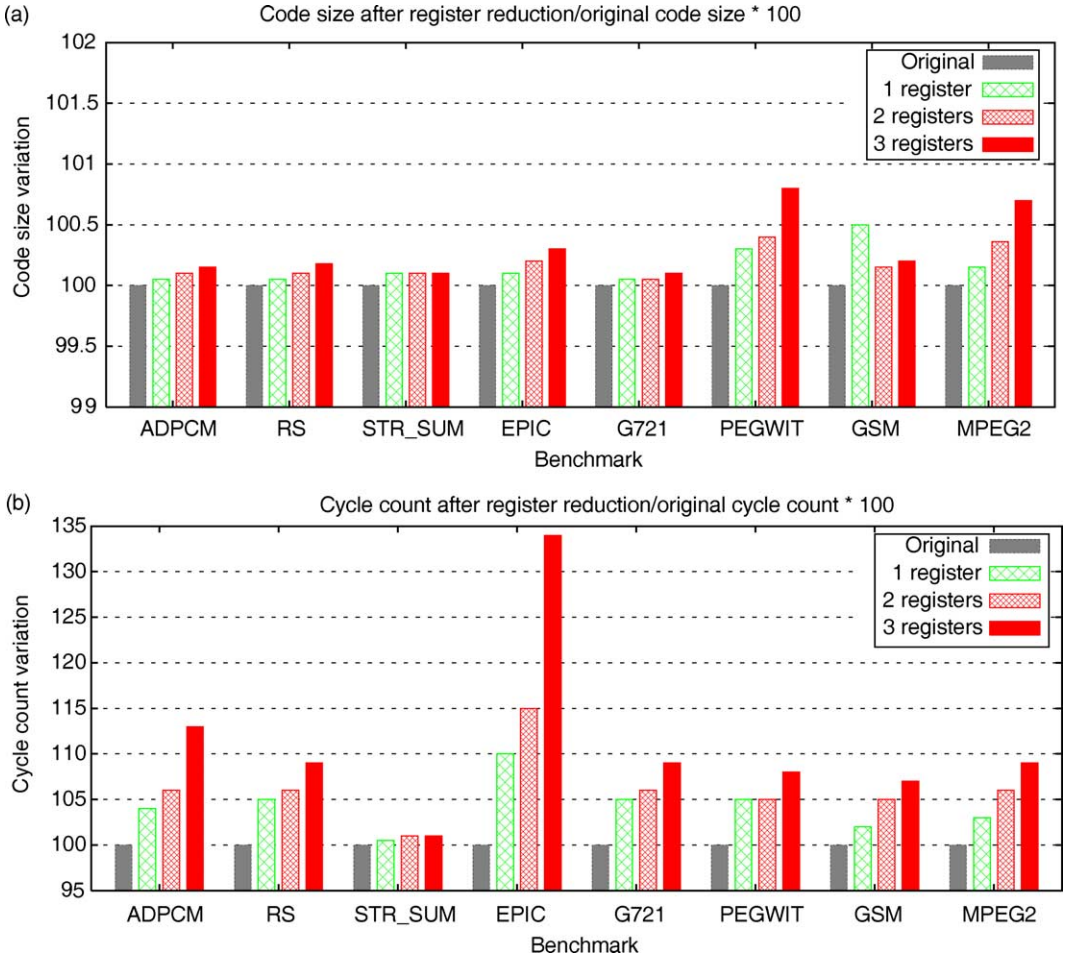

Fig. 6. Effects of register reduction in ARM. (a) Code size increase when the number of available registers are reduced by one, two and three. (b) Cycle count increase when the number of available registers are reduced by one, two and three.

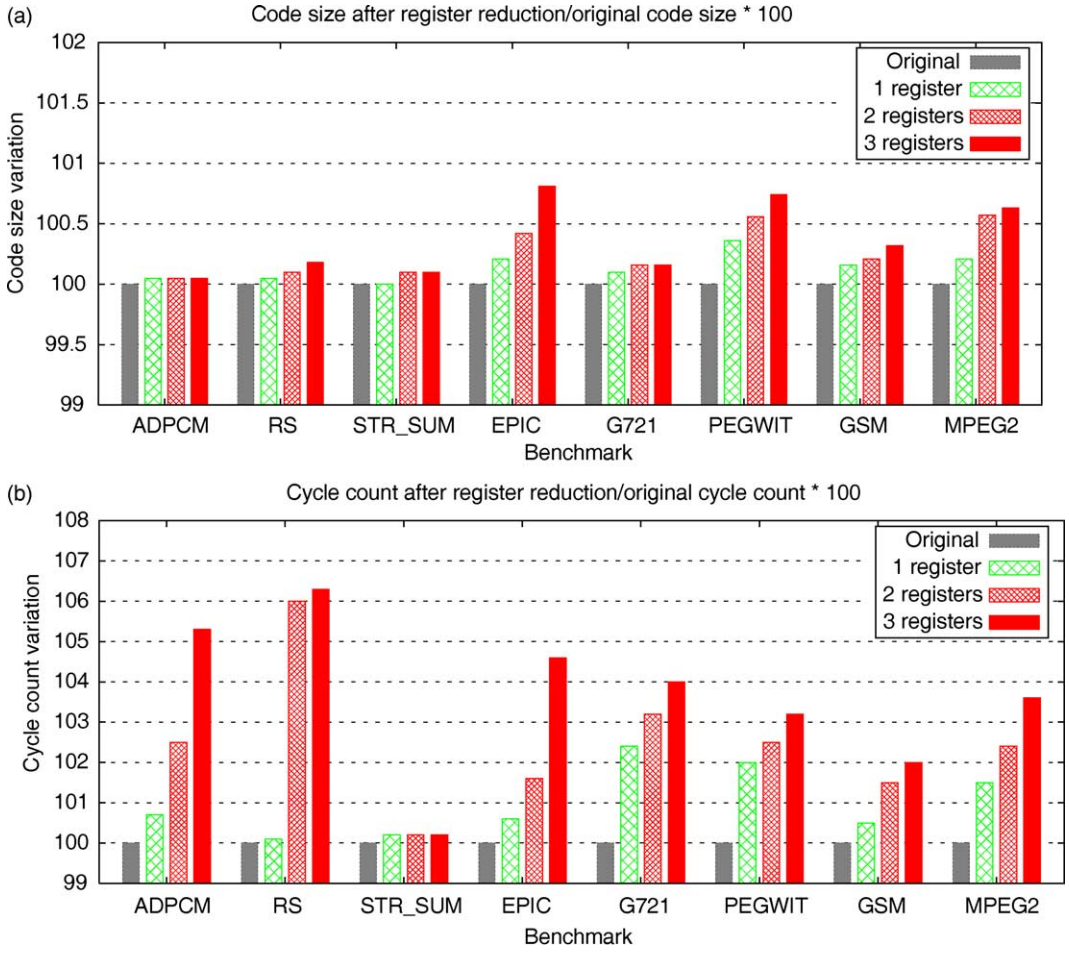

Fig. 7. Effects of register reduction in MCORE. (a) Code size increase when the number of available registers are reduced by one, two and three. (b) Cycle count increase when the number of available registers are reduced by one, two and three.

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, 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.

3. Experimental results

We implemented the proposed register allocation method using GCC and performed experiments for ARM, MCORE and PowerPC (PPC). ARM and MCORE are selected because they are typical embedded processors. PPC is selected to show the effectiveness of the proposed method when a target processor exploits instruction-level parallelism. In the experiments with PPC we use a model of PowerPC 604, which is a four-issue superscalar processor. To measure the cycle count variation, instruction-set simulators (ISSs), ARMulator [14], a MCORE ISS [15], and a PowerPC simulator, PSIM [16], are used. As listed in Table 1, benchmark programs used in the experiments are MediaBench [12] and DSPstone [13]. Some of the programs, DOT_PDT, FIR2DIM, MAT_MUL and STR_SUM use pointers heavily and are included to show that the proposed method can handle pointers appropriately.

First, we performed experiments to determine the number of promotion registers. We generated executables by restricting the number of registers. Figs. 6–8 show the effects of register reduction on the code size and cycle count for ARM, MCORE and PPC, which are obtained with a modified GCC. Based on the results in Figs. 6–8, we can determine the number of promotion registers. The amount of code size increase is less than 1% for the three 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

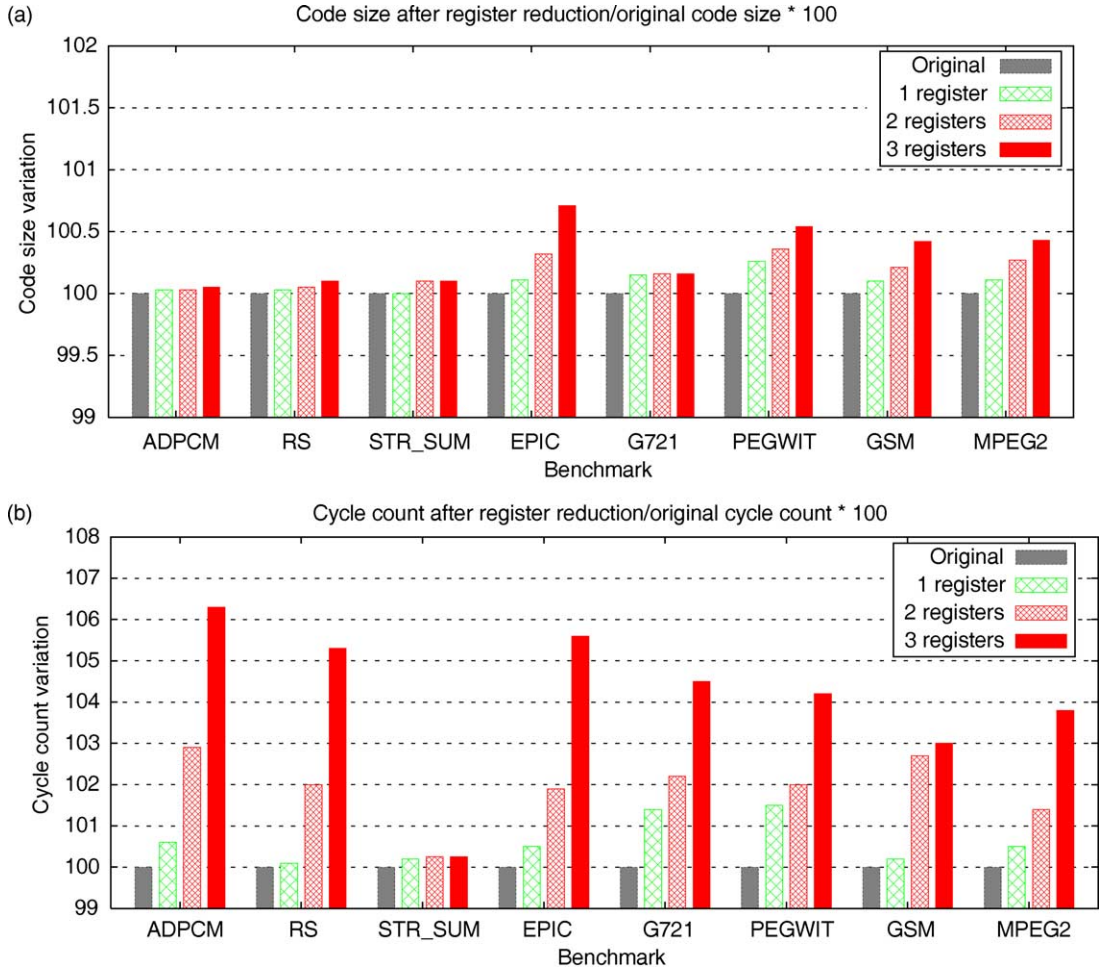

Fig. 8. Effects of register reduction in PPC. (a) Code size increase when the number of available registers are reduced by one, two and three. (b) Cycle count increase when the number of available registers are reduced by one, two and three.

registers as promotion registers in ARM does not impose great penalty on code size and cycle count. For MCORE in Fig. 7, 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. In PPC two promotion registers are reserved for the register promotion because the cycle count increases steeply when the number of registers is reduced by three.

We compiled the benchmark programs using the implemented compiler for ARM, MCORE and PCC 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 applying the proposed register promotion only to the selected functions, the compilation time is reduced as shown in Fig. 9. For MPEG2 the compilation time is reduced to a half of the time when the proposed method is applied to all the functions. The overhead of the proposed method in compilation time is shown in Fig. 10 where the overhead is calculated by excluding profiling time. Profiling time is excluded because it is dependent on the input data. For STR_SUM and DOT_PDT the compilation time becomes two times longer after the proposed method is applied. For other benchmark programs, PEGWIT, ADPCM, MPEG2, G721, and GSM, the overhead is less than 80%. The amount of overhead can be tolerable in general embedded systems because embedded systems are typically designed as final implementations for dedicated functions.

By executing the generated executables on ARMulator, MCORE ISS and PSIM, the cycle counts are measured.

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_PDT	1
FIR2DIM	1
MAT_MUL	1

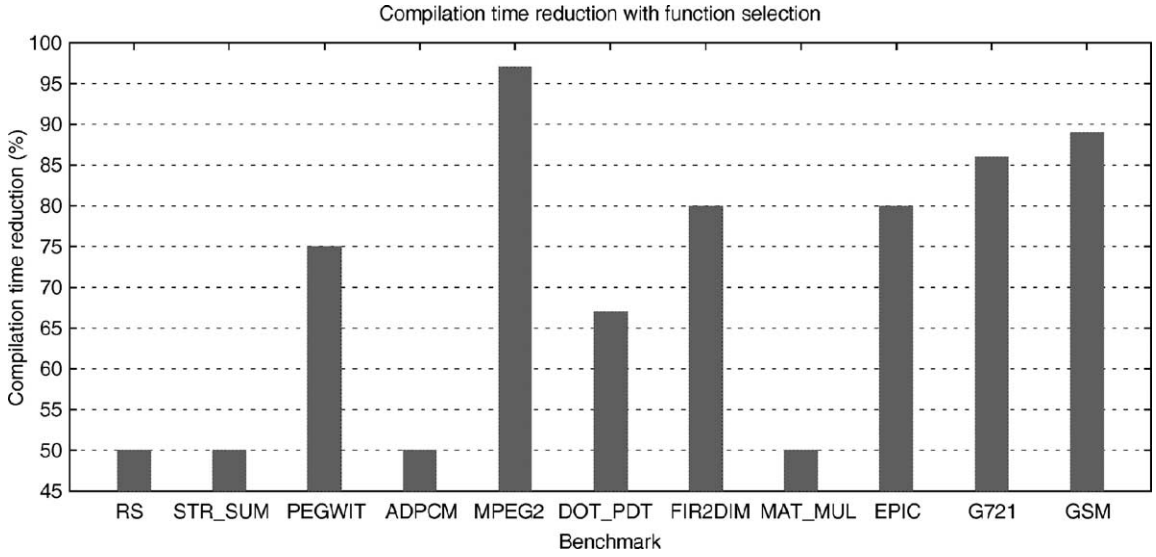

Fig. 9. Compilation time reduction with function selection. The compilation time is reduced by selectively applying the proposed register promotion to the heavily executed functions.

Fig. 11 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 cases of ARM and PPC. However, the cycle count is reduced for all the processors. The average cycle count reductions are about 14, 18, and 9% for ARM, MCORE and PPC, respectively. The maximum cycle count reductions are 30% and 35% for ARM and MCORE, respectively. For PPC the maximum cycle count reduction is about 20%. Fig. 11 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, 20 and 11% for ARM, MCORE, and PPC, respectively.

4. Conclusions

In this paper, we have presented a register allocation technique to enhance performance by promoting memory accesses to register accesses. In the proposed 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.

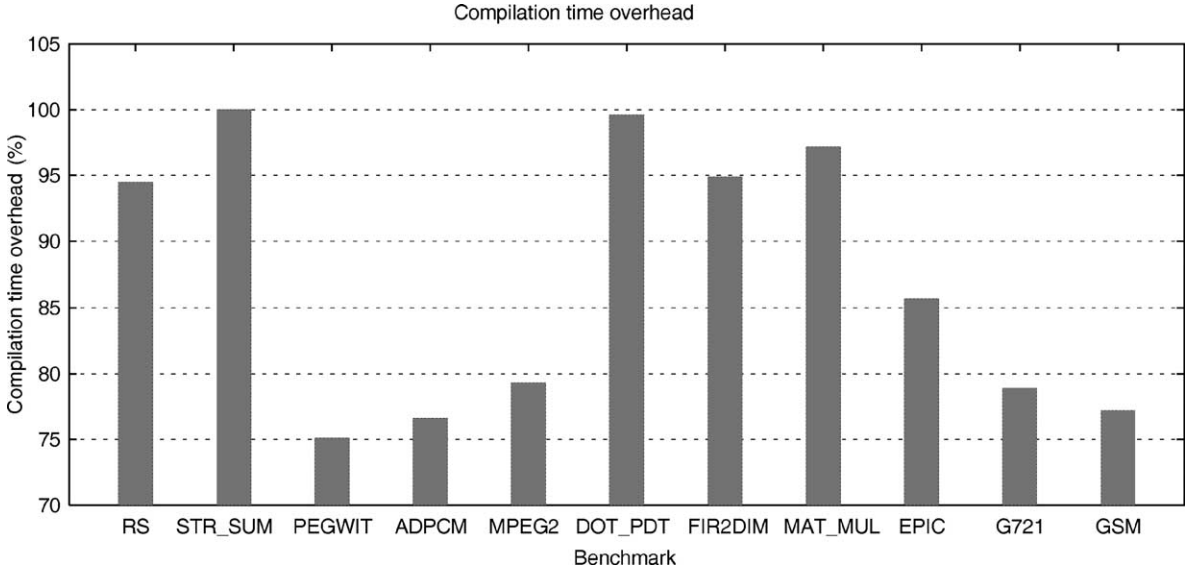

Fig. 10. Compilation time overhead excluding profiling time.

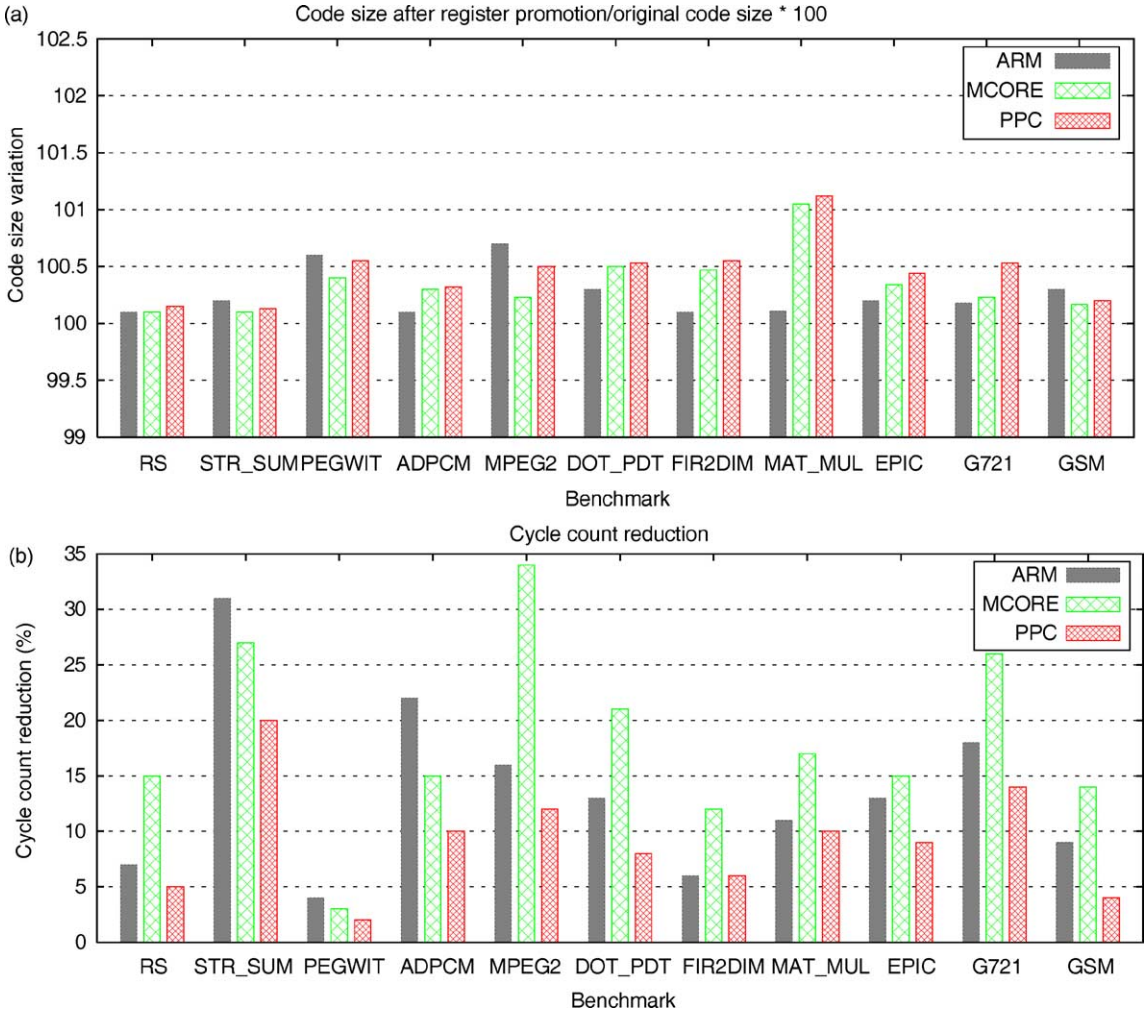

Fig. 11. Code size variation and cycle count reduction after the proposed register allocation. (a) Code size variation. (b) Cycle count reduction.

The experimental results show that the proposed method is effective in that average cycle count reduction is about 14% with increasing less than 1% of the code size.

References

[1] C. Liem, T. May, P. Paulin, Instruction-set matching and selection for DSP and ASIP code generation, Proceedings of the European Design and Test Conference (ED & TC), 1994.

[2] G. Araujo, S. Malik, Optimal code generation for embedded memory non-homogeneous register architectures, Proceedings of the Eight International Symposium on System Synthesis, 1995.

[3] S. Liao, S. Devadas, K. Keutzer, S. Tjiang, G. Araujo, S. Malik, Instruction selection using binate covering for code size optimization, in: International Conference on Computer-Aided Design (ICCAD), 1995.

[4] A. Sudarsanam, S. Malik, Memory bank and register allocation in software synthesis for ASIPs, in: International Conference on Computer-Aided Design (ICCAD), 1995.

[5] S. Liao, S. Devadas, K. Keutzer, S. Tjiang, A. Wang, Storage Assignment to decrease code size, ACM Trans. Program. Lang. Sys. 18 (3) (1996) 186–195.

[6] R. Leupers, P. Marwedel, Algorithms for address assignment in DSP code generation, in: International Conference on Computer-Aided Design (ICCAD), 1996.

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

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

[9] M. Hind, Pointer Analysis: Haven't We Solved This Problem Yet? in: PASTE '01.

[10] R.P. Wilson, M.S. Lam, Efficient Context-Sensitive Pointer Analysis for C Programs, in: SIGPLAN PLDI '95.

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

[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, C. Schlager, DSPstone: a DSP-oriented benchmarking methodology, in: Proceedings of the International Conference On Signal Processing and Technology, Dallas, October, 1994.

[14] ARM Limited. ARM Technical Support FAQs—ARMulator [Online], Available: http://www.arm.com/support/ARMulator.html.

[15] Freescale Semiconductor. Freescale Design Tools Library [Online], Available: http://www.freescale.com/webapp/sps/library/tool_libs.jsp.

[16] Andrew Cagney, PSIM—Model of the PowerPC Architecture [Online], Available: http://sources.redhat.com/psim/.