applied sciences

MDPI

Article

Modeling the Power Consumption of Function-Level Code Relocation for Low-Power Embedded Systems

Hayeon Choi †, Youngkyoung Koo † and Sangsoo Park *,†

Computer Science and Engineering, Ewha Womans University, 52 Ewhayeodae Rd., Seodaemun District, Seoul 03760, Korea; hayeon.choi@ewhain.net (H.C.); kooyoungkyoung@ewhain.net (Y.K.)

* Correspondence: sangsoo.park@ewha.ac.kr; Tel.: +82-2-3277-6643

† International Symposium on Computer, Consumer, and Control (IS3C'18): #1143.

Received: 3 April 2019; Accepted: 4 June 2019; Published: 8 June 2019

Abstract: The problems associated with the battery life of embedded systems were addressed by focusing on memory components that are heterogeneous and are known to meaningfully affect the power consumption and have not been fully exploited thus far. Our study establishes a model that predicts and orders the efficiency of function-level code relocation. This is based on extensive code profiling that was performed on an actual system to discover the impact and was achieved by using function-level code relocation between the different types of memory, i.e., flash memory and static RAM, to reduce the power consumption. This was accomplished by grouping the assembly instructions to evaluate the distinctive power reduction efficiency depending on function code placement. As a result of the profiling, the efficiency of the function-level code relocation was the lowest at 11.517% for the branch and control groups and the highest at 12.623% for the data processing group. Further, we propose a prior relocation-scoring model to estimate the effective relocation order among functions in a program. To demonstrate the effectiveness of the proposed model, benchmarks in the MiBench benchmark suite were selected as case studies. The experimental results are consistent in terms of the scored outputs produced by the proposed model and measured power reduction efficiencies.

Keywords: function-level code relocation; prior relocation-scoring; source code insertion; code profiling; low-power; embedded systems

1. Introduction

The rapid development of hardware and software technologies has resulted in embedded systems undergoing significant changes. The size of the hardware is decreasing, and the software is becoming more sophisticated to meet users' various requirements. However, these changes are made concurrently, but because they do not develop at the same rate, it is necessary to overcome the problems that occur by using a cross-layer approach. In other words, devices of which the compactness has recently been optimized to maximize portability are usually equipped with batteries rather than with continuous power supplies, implying battery-dependent operation. Therefore, systems mounted on these devices must be designed to operate for extended periods of time to meet the requirements of embedded systems, as well as to consider the physical limits such as size, weight, or battery capacity.

Power management techniques that are suitable for application to systems requiring low power such as embedded systems can be divided into several categories. As an embedded system generally consists of microprocessors and peripherals, we classify the techniques into two categories based on the hardware architecture of the embedded system. First, existing low-power techniques such as dynamic voltage and frequency scaling (DVFS) in combination with Power Modes can be applied to power an embedded system, i.e., a method for controlling the microprocessor, which plays the most important role in an embedded system. Second, microarchitectural technology can be used to reduce the energy consumption of specific components, i.e., peripherals other than the microprocessor, such as the main, volatile, and non-volatile memory of the embedded system.

The first category, DVFS, is a technology that varies the voltage or frequency of a computing system according to its performance and power requirements. The technique reduces energy consumption by regulating the voltage supplied to the processors of most computer systems comprising CMOS circuits. Because the power consumed is proportional to the square of the supply voltage, reducing the voltage is a highly effective way to reduce energy consumption. Some commercial microprocessors support DVFS technology to reduce power consumption. Hua et al. studied methods that optimized the voltage level to implement DVFS on a multi-voltage system to save energy in embedded systems. However, the support of multiple voltage levels results in multiple types of overhead, such as power and transition overhead. Solving these problems associated with a multi-voltage system in which such constraints exist would require determining the number of levels and values that would need to be implemented in a multi-voltage system to maximize energy saving. The ideal number of voltage levels that would allow the system to be operated in an energy-efficient manner when compared to an ideal system, that is, a system of which the voltage could be varied randomly, was experimentally proven to be either three voltage levels (2.4 V, 3.0 V, and 5.0 V) or four voltage levels (1.5 V, 2.4 V, 3.3 V and 5.0 V) [1]. Quan et al. proposed two DVFS algorithms for energy saving in a real-time embedded system. The first algorithm determines the minimum constant speed that can be applied while running the entire task set such that the processor can be shut down when idle. The second algorithm generates a constant speed and schedule according to which the voltage is varied to minimize energy consumption. The second of these two algorithms was shown to always save more energy than the first [2].

The second category, microarchitectural technology, utilizes changes in the application properties or workload to dynamically reconstruct the system components to reduce the energy consumed by specific components of the embedded system. Yang et al. studied technology to reduce the energy used by the main memory in embedded systems. The technology uses software-based RAM compression to increase the effective size available for memory usage. Memory compression is only used for applications that can benefit in terms of performance or energy. For these applications, the compression of memory data and swapped-out pages is performed online, thereby dynamically adjusting the size of the compressed RAM area [3]. Steinke et al. proposed a technique for saving energy in an embedded system based on a scratchpad memory. This technique inserts a copy function into the application to copy the set of basic blocks that are frequently used at compile time. Then, at a predetermined point in the flow of program execution, their technique copies a portion of the program from the scratchpad and then runs the program on the scratchpad itself. This saves energy by reducing access to the cache [4].

As mentioned above, studies concerned with the implementation of low-power embedded systems have mainly been proceeding in two directions. The first approach entails applying an existing low-power technique to the embedded system together with a method to reduce the power consumed by the memory installed in the embedded system. Particularly, a study aimed at improving the efficiency of the built-in memory was conducted to reduce the power consumption of the memory itself. It is difficult to conclude whether this approach is suitable for an embedded system in which various types of memories exist because it focuses on the characteristics of one type of memory rather than optimizing the power characteristics of the various memories. This prompted our research aimed at developing a method to drive an embedded system with low power by considering the characteristics of the various types of memory.

Our previous related work involved the use of code migration methodology based on function relocation, to be integrated with the design of low-power embedded systems. These studies changed the location where the program code is loaded from the initially deployed non-volatile memory to volatile memory. The implementation of this migration methodology with low-power embedded systems was demonstrated and their applicability to a wide variety of areas was proven [5–8].

In the current work, we investigated the extent to which the power efficiency, i.e., the degree of power reduction, is influenced by the function-level code relocation based on program instructions. An instruction, which is defined as an atomic unit of a program, is classified into several functional groups, depending on their respective roles. Thus, we developed methods for advanced code profiling aided by a source code insertion technique to determine the effect of grouped instructions on function relocation. Furthermore, we established a model for prior relocation-scoring to guess the order of relocation effects among several functions in one program. Benchmark case studies also confirmed that the function relocation efficiency and the relocatable score of the model exhibit similar trends. The result allows us to estimate the order of influence by relocating a function from non-volatile to volatile memory by determining the ratio of grouped instructions only. Furthermore, it provides the most effective and precise criteria for selecting a function for relocation.

The paper is structured as follows: Section 2 provides the code migration methodology on the basis of functions placed in volatile memory. Section 3 presents an in-depth description of both of the two experimental environments that were used to analyze the influence of the instruction group. The code profiling is presented in Section 4 together with a description of the experiments based thereupon. The power consumed by the grouped instructions was measured to determine the impact of each of the groups on the relocation of function-level code. In Section 5, all the results are integrated in the model for prior relocation-scoring that predicts the order of effects when a function in the program relocates to volatile memory. The validation of the model, which was also demonstrated by conducting several benchmark case studies, is also described in Section 6. The paper is concluded in Section 7.

2. Code Migration Methodology Based on Function Relocation

A microcontroller unit (MCU) can be described as a small computer composed of microprocessors. This unit stores data from a plurality of memories, i.e., heterogeneous memory. These memories execute and load program code or instructions and include peripheral devices for external input and output. In general, embedded systems need to select a suitable microprocessor to meet the complexity and variety of system operations that must comply with various requirements. In general, embedded systems need to select appropriate microprocessors to comply with diverse requirements in terms of various and complex system operations. For instance, microprocessors with the ability to enable wireless connections are necessary for devices that need to communicate with a profusion of other devices. In this respect, when designing microprocessors as the primary method of meeting the various requirements of the system, a method of design that involves mounting the functions that meet the requirements was primarily considered. However, to implement an optimized embedded system, it is necessary to consider not only the microprocessor but also other components that consume power. In particular, memory devices, which are known to consume approximately 60% of the total amount of power consumed by the system, also need to be considered [9].

Typical embedded systems select and apply heterogeneous memory architecture because the usage is applied differently depending on the characteristic of each memory device. Non-volatile memory protects essential contents such as read-only data and program code because of its ability to preserve content even without a continuous power supply. In contrast, volatile memory is mainly used to store data used during program execution and consists of stacks, heaps, data, and code segments. Hence, the program code and data are separated among the heterogeneous components comprising the memory architecture. Thus, most embedded systems select and effectively utilize the most suitable memory device according to the intended usage. Moreover, even if the same operation is performed, it affects the power consumption of the system depending on the type of memory selected. SRAM, which is volatile memory, attempts to reduce the power consumption of the entire system by executing a part of the program code in volatile memory. This is based on the fact that the power required for program execution is less than that of Flash, which is non-volatile memory. This approach was followed in previous studies, which suggested a method of code migration based on function relocation to reduce the power consumption of the entire system by altering the location of the program code. The flow of the methodology is shown in Figure 1.

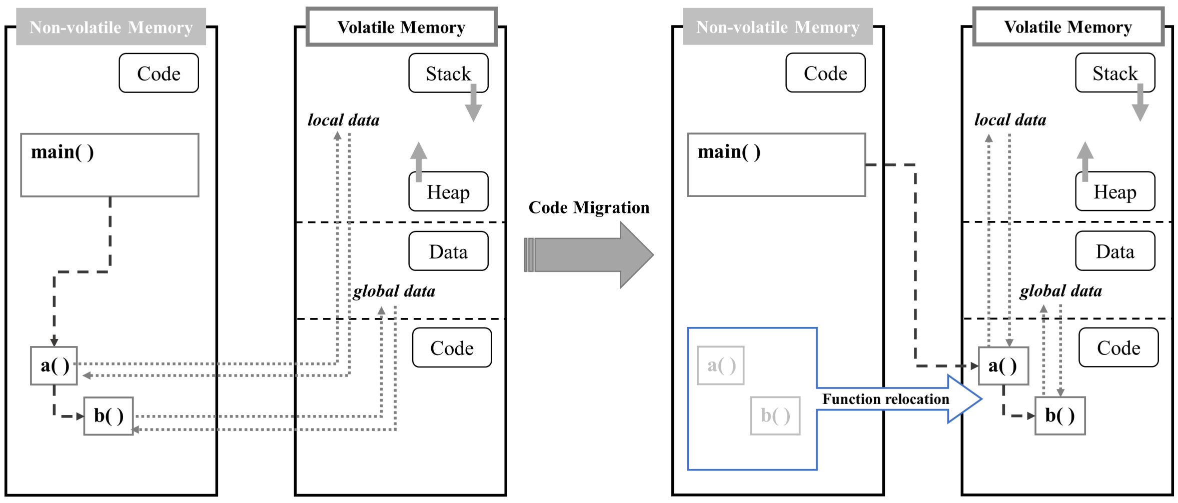


Figure 1. Basic components and processes of code migration methodology.

Unlike non-volatile Flash memory into which program code is loaded, previous studies attempted to reduce power consumption by placing a function in volatile SRAM. In this respect, relocating code involves changing the program code to achieve power reduction by taking into account the operational characteristics of memory by transferring the code of functions from flash memory to SRAM at the function level. We calculate the effect of code relocation by using Equation (1). Defined as the Power Reduction Efficiency (*PRE*), it is the ratio of reduced power as a result of function placement in SRAM instead of in Flash.

$$PRE_f^A = \frac{Power^{non-migration} - Power^{migration}}{Power^{non-migration}} \times 100, \tag{1}$$

where A is an application, i.e., a program loaded on an experimental board, and f is a function, i.e., a function in an application. Further, $Power^{non-migration}$ is the power consumption of applications used in the normal way, and $Power^{migration}$ is the power consumed when the code migration methodology is used on the basis of function relocation.

This result indicates that the increased time complexity of the program selected for experimentation has an effect on the increase in the power reduction efficiency of function relocation. Furthermore, the efficiency varies depending on the segment of the volatile memory that is accessed [5–8]. These results confirm the advantageous effect of the proposed function-level code relocation to reduce the amount of power consumed. The analyses conducted in previous studies were comparatively easy to achieve because they were based on a human-understandable programming language, yet they were limited to an abstract analysis based on experimental data. An in-depth analysis, however, requires substantial analysis based on actual program execution at the lower language level. Thus, in this work, we investigate effective program execution at the lower level of a language, e.g., at the instruction level, and attempt to observe the influence of code migration based on function-level code relocation on the rate of power reduction.

3. Configuration of Entire Environment

Programmers are able to create programs and design systems by using the following formal syntax of programming languages built for convenience. However, the system is not designed to execute the programming language itself. Because the system does not understand a high-level language, it needs to be transformed via several steps such as in Figure 2. In the general flow of transformation, the language human beings can understand is assembly language. This language strongly coincides with machine language, which is the only language a computer understands. In this respect, analysis using assembly language, which is defined by instructions, is the ultimate and essential way to understand the execution of a program accurately.

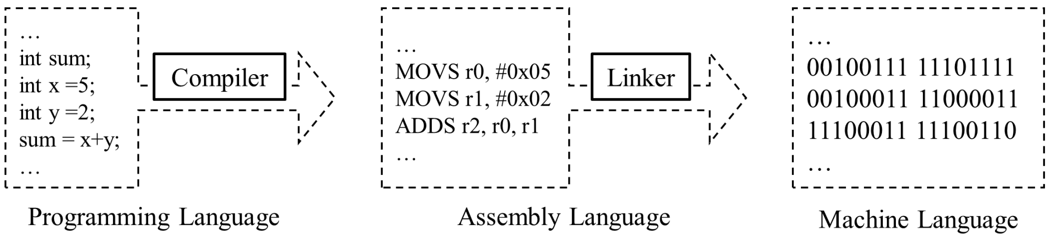


Figure 2. Flow of language transformation.

Ultimately, the analysis of the effect of instructions on the power consumption of a system requires an environmental setting that can measure the power consumed by a particular MCU. In other words, the implementation of low-power embedded systems can be demonstrated by accurately measuring the power consumption in the actual environment. In this study, the environment is simultaneously configured for conducting the instruction trace and also the power measurement with high accuracy. Details of the environment appear in Table 1.

Table 1. Environment configurations.

Component		*Environment*	
		Instruction Trace	**Power Measurement**
Hardware	MCU	MSP432E401Y [10]	MSP432P401R [11]
	Debugger	Keil ULINK*pro* [12]	TI XDS-110 [13]
Software	Compiler	ARM CC Compiler [14]	TI Compiler [15]
	Instruction Set	ARM Instruction Set [16]	ARM Instruction Set [16]
	IDE	Keil uVision5 [17]	Code Composer Studio [18]
	Method	ETM Trace [19]	EnergyTrace^{TM} [20]

Compared to other computing systems, environmentally sensitive embedded systems should not only examine the software tools thoroughly but also hardware related devices or peripheral components. All these components influence our understanding of the overall performance and stability of the systems. In this regard, tracing the instructions and measuring the power of the actual system reflect the actual operations. Consequently, these processes create firm grounds for the subsequent results of code profiling and show the model for prior relocation-scoring to be reasonable.

3.1. Instruction Trace

Instruction trace is one of the tracking methods employed to extract memory transactions and instructions from the MCU. This method makes it possible to step through and debug code execution, a process which is useful for analyzing the performance of software. Rather than being a set of assembly codes directly translated from a programming language, the trace method is the sequential collection of actual executing instructions. The combination of instructions, i.e., instruction trace description, therefore, enables the complete flow of a program to be understood.

The MSP432E401Y Launchpad manufactured by Texas Instruments (TI) is a high-performance ARM Cortex-M4F-based microcontroller that uses a 32-bit ARM instruction set. In addition, this MCU combines 1024 KB Flash memory, 256 KB SRAM, and 6 KB EEPROM. A distinctive feature of the MCU is that it includes GPIO ports that enable the use of Embedded Trace Macrocells (ETM). All the information tracked via the ETM buffer interface is stored by the processor. This information includes all instructions executed and a real-time stream of read/write data. Despite these impressive characteristics, ETM is non-intrusive, i.e., it has no influence on program execution. Hence, using this MCU and the ETM feature is an attractive option to understand the operating content of a program without the effect of external interference. Keil ULINKpro, which is designed for experimentation, has a debugging function and uses a unique technique for trace streaming. In other words, MSP432E401Y Launchpad would have to be combined with the debugger to make use of ETM instruction trace.

In this study, we connected MSP432E401Y MCU and ULINKpro to construct the tracing environment. ETM required a 20 pin CoreSight connector with a pin size of 1.27 mm. Because this component is not usually installed on the MCU, a custom adapter was necessary, as shown in Figure 3. In addition, five signals sized 2.54 mm, four GPIO pins for data transfer, and one clock pin, were placed on the other side of the custom adapter.

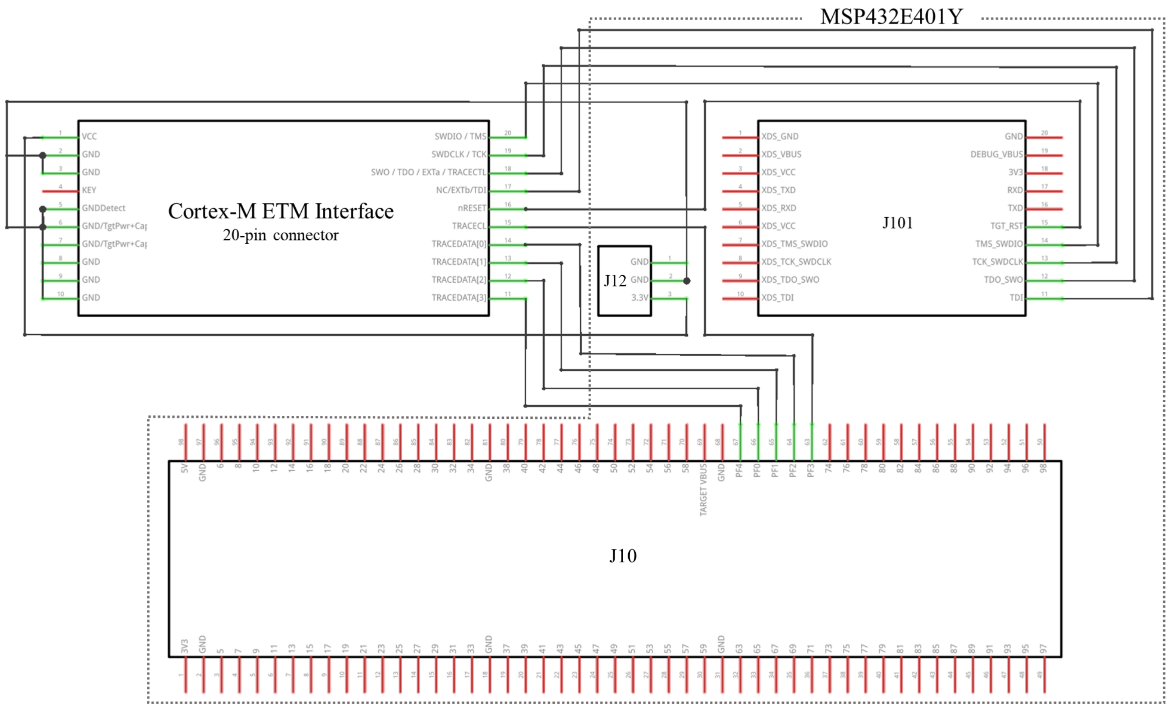


Figure 3. Custom adapter.

The trace results are shown in raw assembly code based on the ARM instruction set. All groups were restructured to conduct further in-depth analysis, and abbreviations were prepared based on the instruction group defined in the ARM instruction set. Among these, the instructions included in the "Multiple Register Load and Store" group access memory require several general-purpose registers. The "Register Load and Store" group could include the aforementioned group in a large sense. Furthermore, as the ARM operating modes are switched by using the "Status Register Access" and "Coprocessor" groups, they are not considered. Table 2 lists all of these redefined instruction groups.

Table 2. Group of ARM instruction set.

Instruction Group	*Redefined Instruction Group*	*Example*
Data Processing	DP Group	ADDS, SUB, ORR, MOV,...
Register Load and Store	RLS Group	LDR, STR, PLD, SWP,...
Multiple Register Load and Store		LDM, STM, PUSH, POP,...
Branch and Control	BC Group	B, BL, BX, BLX, ...
Status Register Access	Not Considered	MRS, MSR, CPS, SETEND,...
Coprocessor	Not Considered	CDP, MCR, MRC, LDC, ...

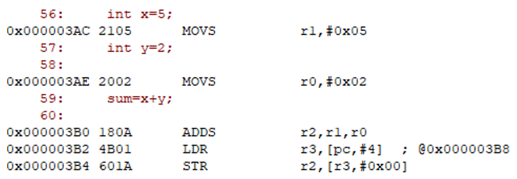
As mentioned above, an instruction trace does not constitute a translated assembly language set, but an exact trail resulting from the original execution of a program. Therefore, preprocessing is inevitable prior to further analysis.

- In this study, a unit selected for relocation is a single function in a program. An effective analysis requires the instruction trace result to be separated at the function level.
- After dividing the result by functions, every instruction in a function is counted depending on the redefined instruction groups in Table 2. This step allows us to grasp the scale of each group.
- Some instructions in the RLS group additionally need particular attention for correct counting, e.g., multiple memory accesses.
- Preprocessing, as described above, based on Python 3.6, concludes the instruction trace description as a result.

3.2. Power Measurement

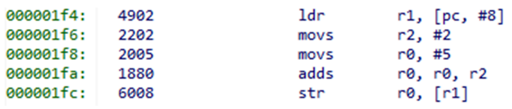
The MSP432P401R Launchpad, which is used to measure power, was also designed by TI. The processor is the same as the Launchpad used in Section 3.1 and has a heterogeneous memory component, 256 KB Flash memory, and 64 KB SRAM, which can be used to implement function-level code relocation. Moreover, unlike other microcontrollers, this is specifically designed for low-power operation. In particular, the EnergyTrace+ hardware, which enables the implementation of the EnergyTrace^{TM} technique used to generate the estimated measurement summaries, such as the amount of power consumed, the amount of energy used, and the life of the battery, is included. The primary factor of the measurement is power consumption while implementing a low-power embedded system is the ultimate objective. Any electrical signal such as power relies on the ambient environment. Thus, all connections in the Launchpad except for the essential components are separated to allow precise measurement.

When using instruction trace, i.e., as described in Section 3.1 and power measurement, as explained in Section 3.2, the environments used differ in several respects. The MSP432E401Y Launchpad used for instruction trace supports the creation of programs using Keil uVision5 to trace instructions through the ARM CC compiler. Meanwhile, the MSP432P401R Launchpad, which is used for power measurements, supports the creation of programs that use the TI compiler to measure power consumption using Code Composer Studio (CCS). However, both MCUs are designed with the ARM Cortex-M4F microprocessor based on the same ARM instruction set. This indicates that the instruction trace description, which is the result of the instruction trace, is correlated with the measurement summary, which is the power measurement result. Figure 4 presents a further comparison of the executed codes.



```
    56:     int x=5;
0x000003AC 2105      MOVS         r1,#0x05
    57:     int y=2;
    58:
0x000003AE 2002      MOVS         r0,#0x02
    59:     sum=x+y;
    60:
0x000003B0 180A      ADDS         r2,r1,r0
0x000003B2 4B01      LDR          r3,[pc,#4]  ; @0x000003B8
0x000003B4 601A      STR          r2,[r3,#0x00]
```

(a) Instruction trace.



```
000001f4:   4902          ldr     r1, [pc, #8]
000001f6:   2202          movs    r2, #2
000001f8:   2005          movs    r0, #5
000001fa:   1880          adds    r0, r0, r2
000001fc:   6008          str     r0, [r1]
```

(b) Power measurement.

Figure 4. Executed assembly codes in both environments.

The disassembly window captured from each environment implies the execution of each respective sample of programming language shown in Figure 4. As shown, the instructions included in these samples are highly comparable, despite the environments using different compilers. While the two compilation environments differ in numerous ways, such as the type of registers or expressed syntax, an important factor affecting the description of instruction trace is the type of operation the instruction performs, e.g., MOVS, ADDS, LDR, and STR. Additionally, the number of instructions from each group of instructions is the same, even if the environment used is different. Hence, concerns about the inability to maintain consistency that may arise from the use of different environments can be eliminated. In other words, extraction of the instruction trace description from the instruction trace environment and analyzing the power summary results in the power measurement environment are valid forms of analysis.

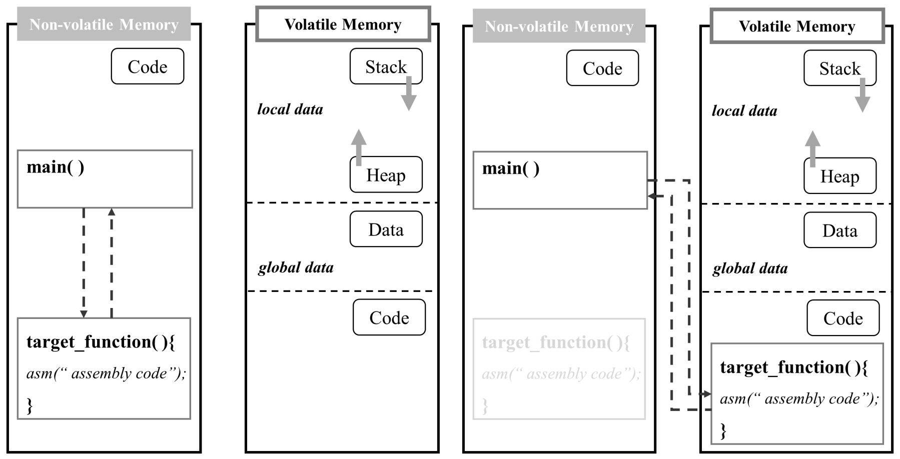
4. Code Profiling Using the Source Code Insertion Technique

The program comprises various instructions, all of which are one of three types of redefined instructions. Because all groups structurally proceed along unique paths in the ARM architecture datapath, they perform distinct operations and affect the system in different ways. In addition, each group of instructions consumes a different amount of power, which allows us to distinguish between the effects on the system in terms of power. This study extends the existing code migration methodology based on function relocation, which is based on the fact that power consumption may vary depending on the instruction groups. Thus, there exist power differences between instruction groups, and these differences have a high probability to be huge if instructions are placed in non-volatile memory. This section explains the use of code profiling to reveal the influence of instruction groups. During the process, non-complicated source code is inserted into the program, and then the power consumption is measured for the purpose of analysis.

4.1. Design of Code Profiling

An inline assembler is one of the features of a compiler that merges code written in assembly language into a program that already contains a programming language. The feature has the advantage of clearly inscribing the assembly code in the designated position for execution. Unfortunately, it is not offered by all compilers. However, the TI compiler, which is used to measure power consumption, is an exception in that it supports the feature. Therefore, the technique we implemented entailed inserting the source code by using an inline assembler to execute the desired assembly code. The source code insertion was performed during code profiling to interpret the power consumed by the instruction groups for the different memory devices.

The target function is placed in Flash memory without any particular modification, or is relocated via modification of the program code and placed in the SRAM. This section describes our intention to examine whether the relocation of function-level code enables the power consumption to be reduced depending on the instruction groups. Thus, as shown in Figure 5a,b, experimental and control groups are specified. In both of these cases, the target function contains different assembly language depending on the instruction groups, and the following experimental examples of instructions are performed on each of the instruction groups.



(a) Control group. (b) Experimental group.

Figure 5. Code relocation scheme using assembly code injection technique.

- DP group: The group comprises a set of instructions that execute data processing operations in logical arithmetic units, and examples in which a set of instructions are configured are presented in Table 2. We used code profiling to define the most typical addition operation in the target function, i.e., the ADDS instruction, to validate the effect of simple arithmetic operations.
- RLS group: The instructions to access memory for the transmission of data belong to this group. Because it has a heterogeneous memory structure, the same experiment was conducted by separating the address of the target memory to detect the apparent influence. Other than this, the experiments were conducted according to the purpose, i.e., read and write, of accessing the memory. The TI compiler, however, does not provide a feature that allows an inline assembler to gain direct access to a specific memory address. We created an additional DP group by combining the MOVW, MOVT, and LDR or STR instructions, to generate the necessary instructions and define the target function.
- BC group: The instructions contained in this group control the flow of the program. As listed in Table 2, instructions such as B, BL, and BX exist. In the same way, a general instruction is selected from the DP group, in which case B is the instruction comprising the target function.

4.2. Result of Code Profiling

Based on 1,000,000 instructions, the number of instructions was increased 10 times and 100 times to perform code profiling, and at the same time, the power measurement step was carried out. Table 3 presents the results of code profiling, and indicates the mean power measured for each instruction group.

Table 3. Power consumption based on code profiling result.

Instruction Group	*Position of the Function*	*Number of Instruction*	*Power Consumption (mW)*
DP	Flash	x	2.997
		10x	3.006
		100x	3.012
	SRAM	x	2.651
		10x	2.656
		100x	2.658
BC	Flash	x	2.999
		10x	3.007
		100x	3.010
	SRAM	x	2.659
		10x	2.661
		100x	2.663

Table 3. *Cont.*

Instruction Group		*Position of the Function*	*Number of Instruction*	*Power Consumption (mW)*
DP		Flash	x	2.997
			10x	3.006
			100x	3.012
		SRAM	x	2.651
			10x	2.656
			100x	2.658
BC		Flash	x	2.999
			10x	3.007
			100x	3.010
		SRAM	x	2.659
			10x	2.661
			100x	2.663
RLS	read Flash	Flash	x	3.364
			10x	3.378
			100x	3.382
		SRAM	x	3.068
			10x	3.077
			100x	3.080
	read SRAM	Flash	x	3.184
			10x	3.228
			100x	3.236
		SRAM	x	2.772
			10x	2.777
			100x	2.780
	store SRAM	Flash	x	3.232
			10x	3.445
			100x	3.515
		SRAM	x	2.771
			10x	2.953
			100x	2.994

In the case of RLS, because the memory segment to be accessed differs according to the type of data to be transferred, additional classification is possible. For example, read-only data marked with the "const" keyword is typically declared in flash memory. Furthermore, static and global data reside in the data segment of the SRAM, and local data and parameters reside in the stack segment of the SRAM. Additionally, the apparent effect of memory access was determined by conducting a total of five experiments on different types of data. These experiments were performed to infer the effect of memory access, i.e., by instructions in the RLS group, according to the data types mentioned above, and the results are shown in Figure 6.

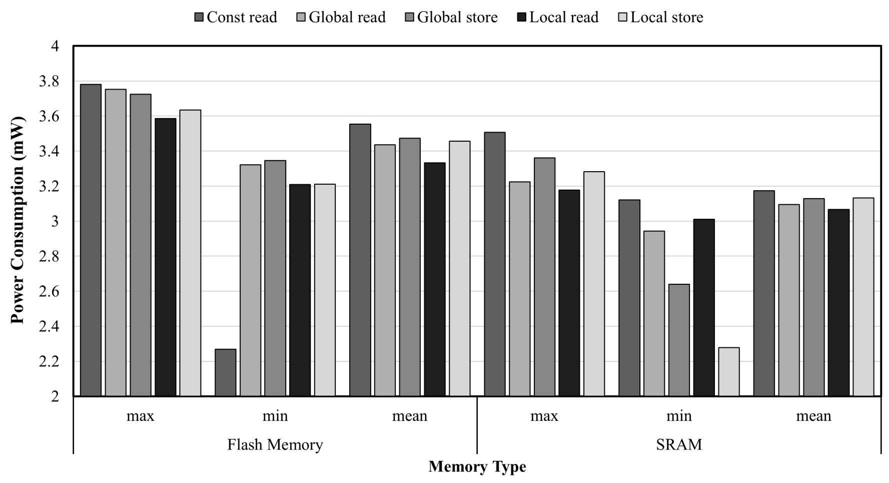


Figure 6. Additional classification of RLS group.

As the number of instructions increased, the power consumption tended to increase. In addition, regardless of function-level code relocation, the target function consumed the most power if it consisted of an instruction of the RLS group, and consumed the least power if it consisted of an instruction of the DP group. Moreover, when the function-level code relocation, i.e., the target function, is located in the SRAM, consistently less power was consumed regardless of the type of instruction group. However, the extent to which power reduction occurs, i.e., the Power Reduction Efficiency (PRE), is different for each instruction group, as shown in Figure 7.

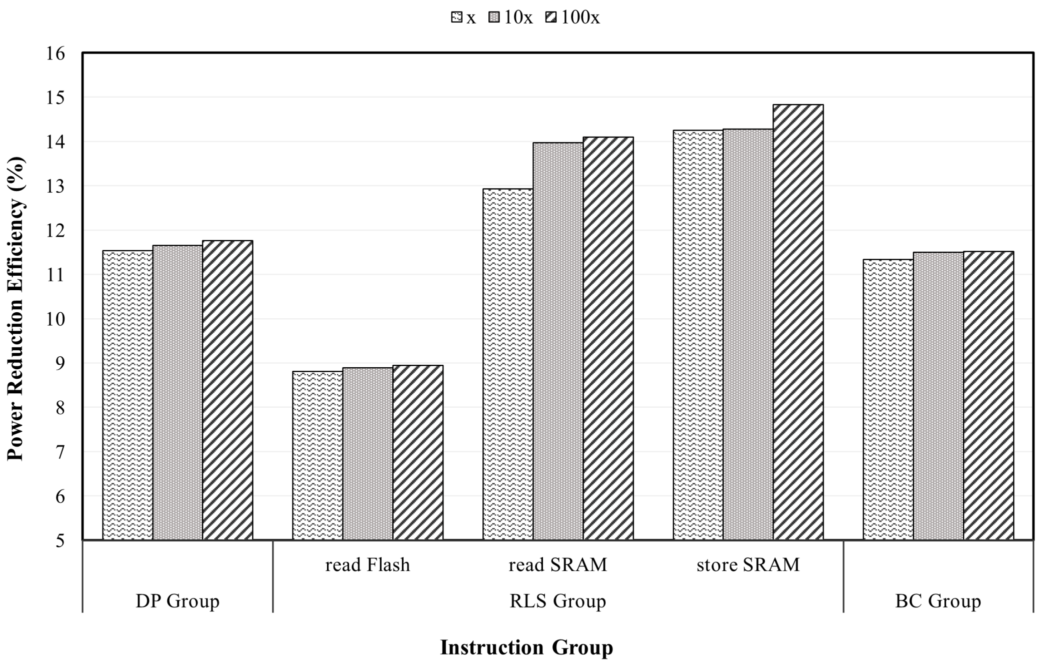


Figure 7. Code profiling result according to instruction groups.

The PRE due to function relocation was the highest for the RLS group of instructions by as much as 12.623%. This efficiency is the average value of the efficiency of "read Flash," "read SRAM," and "store SRAM." In addition, for the DP group, the PRE due to function relocation was the second highest, even though this group consumed the least power. Furthermore, as the number of instructions increased in all groups, the PRE also tended to increase. That is, although the influence of function-level code relocation differed depending on the instruction group forming the target function, it means that the influence increases as the number of instructions increases. This suggests that when estimating the effects of code relocation, not only should the group of instructions be analyzed, but also the effect of the number of instructions. We established a prior relocation-scoring model to reflect all these trends, and this is described in more detail in Section 6.

5. Prior Relocation-Scoring Model

Code profiling using the source code insertion technique demonstrates that function-level code relocation uniquely influences the power consumption according to instruction groups. It also shows that the number of instructions is also affected. In this respect, as the number of instructions that constitutes the target function increases, the impact of the instruction group on function-level code relocation increases; thus, the relationship between the number of instructions and the Power Reduction Efficiency (PRE) should be analyzed. In addition, because the target function consists of a combination of multiple groups of instructions, this analysis should be conducted for all instruction groups. This was achieved by performing regression analysis [21] in which the PRE was set as the dependent variable, the number of instructions were set as independent variables, and by modeling the relationship between these two types of variables. The results are shown in Figure 8.

The dependent variable, PRE, was set as the result of the code profiling in Table 3 and Figure 7 and regression analysis was performed. The slope is the largest in the RLS group and the smallest in the BC group. This is in agreement with the code profiling results. The regression model is considered to be less useful if the coefficient of determination(R^2), which is a measure of the degree to which a model estimated in statistics is appropriate for given data, has a value close to zero.

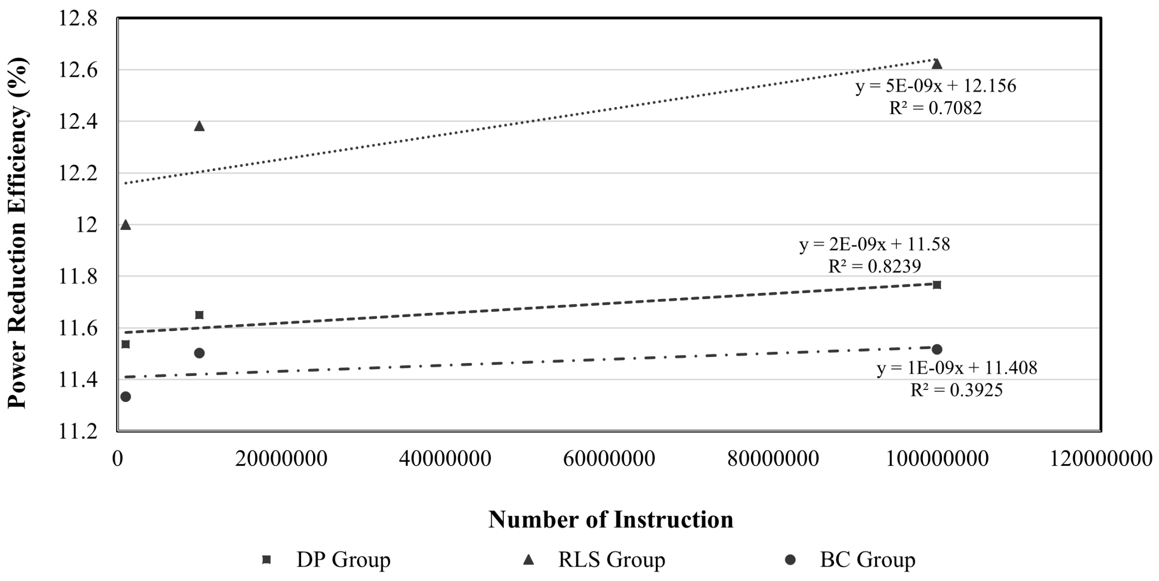


Figure 8. Regression analysis results.

However, as shown in Figure 7, the decision coefficients of all instruction groups have a value greater than zero, indicating that regression analysis is a useful method for analyzing the relationship between the number of instructions and PRE. Based on these results, this study proposes a prior relocation-scoring model using function-level code relocation in the form of Equation (2).

$$RS_f = \sum_{g \in G} \left(N_g \times Slope(g) + Intercept(g)\right) \times R(g)^2 \times r_{N_g}, \tag{2}$$

where N_g indicates the number of instructions of each group and G is the set of ARM instruction sets redefined in Table 2. Further, $Slope(g)$, $Intercept(g)$, and $R(g)$ are the results obtained by regression analysis, as shown in Figure 8, and r_{N_g} is the percentage of instructions in the group.

As the equation implies, the number of instructions and the result of regression analysis are the main factors contributing to the Relocatable Score (RS) calculation. Thus, the instruction trace, as explained in Section 3.1, produces an instruction trace description including N_g and r_{N_g}. The model calculates RS for each function. This enables us to determine the PRE value used in the regression analysis of each function by measuring the power experimentally in the Section 3.2 environment.

The prior relocation-scoring model detects a function that is supposed to be relocated before others. In other words, the calculated RS enables us to identify the order in which functions need to migrate to the other type of memory. Therefore, a function with a high RS score is preferentially relocated, whereas the relocation of a low RS-scored function is more likely to be less effective compared to other functions in a program. The ultimate goal of relocation is to maximize the relocation efficiency, defined as the PRE in Section 2. As the PRE values can only be determined by conducting real experiments, being able to estimate the order of PREs without actual measurement would be meaningful and the low-power expandability of the system could be predicted.

In short, the proposed model assigns a grade to the RS scores to allow the order of the PRE values in a program to be predicted. A unique function intended for earlier relocation is assigned a high RS score for a high PRE value; on the other hand, a function with a low RS score is expected to have a low

PRE value. Each of the RS scores can be calculated by using Equation (2). Consequently, the model for prior relocation aims to obtain a well-predicted relocatable order of functions by applying the result of code profiling and regression analysis. A comparison of RS and PRE reveals whether the model is able to correctly predict the order of prior placement. In conclusion, the RS score possibly establishes a clear standard for determining which function to be allocated to capacity-constrained SRAM without the need to perform the actual measurement.

6. Experimental Results and Analysis

In this section, we evaluate the proposed prior relocation-scoring model by using various benchmark programs. An instruction trace environment was set up to design the model. The instruction trace description that was derived was intended to describe several specifications of functions regarding instruction groups; for example, the number of instructions in each group, the power reduction efficiency of placing an example instruction in the other type of memory, and the absolute power values during execution.

Then, the programs were ported to the power measurement environment for the model experiment. Based on the MiBench benchmark suite [22], which is known as an adequate tool for analyzing embedded processors, MiBench is composed of six categories including Automotive and Industrial Control, Network, Security, Consumer Devices, Office Automation, and Telecommunications. An experiment requires sequential work to be carried out. First, Instruction Trace must be run, after which it is necessary to measure the power factor when applying the proposed methodology. The results are collected by implementing the ETM Trace Method of Keil ULINKpro in the Instruction Trace environment and preprocessed by using Python. Subsequently, for the measurement of the power factor according to the application of the proposed methodology, the function is relocated to the SRAM area. At this time, the size of the SRAM memory on the experimental board is important and, as mentioned, it has a size of 64,000 bytes. In addition, because the SRAM is divided into several segments, Stack, Heap, Data, and Text amongst others, the size of the text area to be used by the function being relocated is limited. Therefore, we excluded some benchmarks that could not be executed because of the hardware limitation, e.g., requirements in terms of peripherals of experimental boards and memory capacity. These results enabled us to list our target applications. Because our model conducts function-level relocation, the functions comprising each program are also provided in Table 4.

Table 4. Benchmarks of MiBench selected as experimental group.

Categories	*Application*	*Function*
Automotive and Industrial Control	Basicmath	solvecubic usqrt
	Qsort	qsort compare swap
Network	Dijkstra	dijkstra enqueue dequeue qcount
Security	Blowfish	init encrypt decrypt f
	SHA	stream init update final transform

The occurrence of code relocation arranges the location of a function in a program. A function is fundamentally located in Flash memory without critical modification, whereas the insertion of unique codes can cause it to migrate to SRAM. In Figure 4, therefore, we separate the cases into two groups: The experimental group and the control group. Each group positions the target function differently. A target function in the experimental group is in SRAM, whereas a target function in the control group is in Flash memory. Because the control group is the standard for experimental comparisons, we specified the power consumption of the group as 100%. The value of the experimental group is subsequently analyzed as the relative power consumption. If the relative power consumption is smaller than 100%, it implies that execution in SRAM required less power than in Flash memory. This evaluation defines a target function as one of the functions in a benchmark program. Therefore, we migrated each function to SRAM one by one to obtain a measurement summary. The results are displayed as bars in Figure 9.

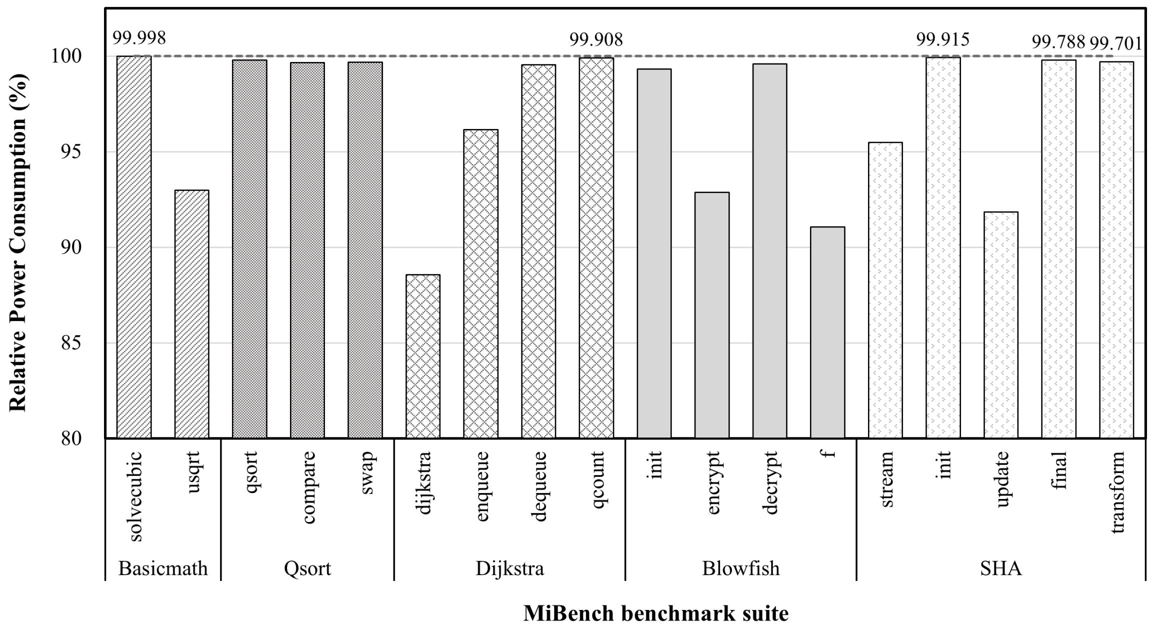


Figure 9. Relative power consumption of benchmark functions.

The methodology for code migration, which is based on the relocation of a function, previously confirmed that, in all cases, the power consumed when the function is placed in SRAM is lower than when it is located in Flash, as shown in Figure 9. The results reveal a clear difference in the extent to which the reduction in power consumption depends on the function that constitutes the application. This is the same trend as the result of the code profiling, which was performed to establish the prior relocation-scoring model. In other words, functions with comparatively low differences in their relative power consumption contain fewer instructions than other functions of the applications. Moreover, this experimental work proved once again that the number of instructions constituting a function should be considered when establishing the model; in addition, it is proven that the application of the code migration method to an actual embedded system enables the system to operate at low power.

For greater clarity, it would be necessary to analyze not only the power consumed but also the energy consumed by an application. The energy consumed by the application is calculated as the product of the voltage, current, and time, with the voltage being proportional to the time and current because it is supplied in equal amounts. At this point, however, the application must compare the amount of energy consumed per unit of time rather than comparing the total amount of energy consumed. The less energy consumed per unit time, the longer the battery lifetime. The results shown in Table 5 confirm that less energy is consumed per unit time when any function is rearranged than when the application is executed without rearranging any functions. In other words, when our code migration methodology based on function relocation is applied, less energy is consumed per unit time than without this methodology. This proves that applying this methodology to a small-sized embedded system that needs to be operated for a few months only, powered by a battery without an external power supply once installed, enables long-time operation of the system because the energy used per unit time is less.

Table 5. Energy consumed per unit second of benchmark functions.

Application	*Applicability of Code Migration*	*Relocated Function*	*Energy Consumed per Unit Second (mJ)*
Basicmath	N	-	2.588
	Y	solvecubic	2.581
		usqrt	2.467
Qsort	N	-	3.401
	Y	qsort	3.390
		compare	3.306
		swap	3.390
Dijkstra	N	-	3.355
	Y	dijkstra	2.957
		enqueue	3.224
		dequeue	3.343
		qcount	3.348
Blowfish	N	-	3.389
	Y	init	3.362
		encrypt	3.135
		decrypt	3.364
		f	3.077
SHA	N	-	3.340
	Y	stream	3.189
		init	3.338
		update	3.330
		final	3.333
		transform	3.068

The validity of the code migration method was demonstrated by conducting an experiment. In addition, the performance of the proposed prior relocation-scoring model was analyzed by using the selected benchmarks for the experimental group in the instruction trace, as mentioned in Section 3. This provided the value of the variables applicable to the model, which are listed in Table 6.

Table 6. Benchmark information.

Application	*Function*	*Number of Instruction (N_g)*			*Percentage of the Instruction (r_{N_g})*		
		DP Group	**RLS Group**	**BC Group**	**DP Group**	**RLS Group**	**BC Group**
Basicmath	solvecubic	55,396	23,426	17,743	0.20035	0.67980	0.21154
	usqrt	221,100	11,034	66,133	0.79965	0.32020	0.78846
Qsort	qsort	15,003	7807	10,986	0.13286	0.06840	0.27635
	compare	74,870	57,454	16,780	0.66301	0.50340	0.42210
	swap	23,051	48,872	11,988	0.20413	0.42820	0.30155
Dijkstra	dijkstra	1,061,012	1,680,350	481,404	0.81304	0.80641	0.70985
	enqueue	201,968	262,800	193,314	0.15477	0.12612	0.28505
	dequeue	42,012	133,675	0	0.03219	0.06415	0.00000
	qcount	0	6920	3458	0.00000	0.00332	0.00510
Blowfish	init	32,438	15,912	6888	0.05442	0.03112	0.04253
	encrypt	257,710	63,585	56,882	0.43237	0.12434	0.35123
	decrypt	33,580	8284	7412	0.05634	0.01620	0.04577
	f	272,316	423,584	90,768	0.45687	0.82834	0.56047
SHA	stream	133,135	99,814	35,031	0.23667	0.21626	0.05320
	init	10	7	12	0.00002	0.00002	0.00002
	update	427,490	360,028	621,307	0.75993	0.78006	0.94353
	final	20	25	21	0.00004	0.00005	0.00003
	transform	1887	1666	2120	0.00335	0.00361	0.00322

Figure 10 shows the correlation between the function placement efficiency PRE and the value of RS derived from the model. Based on the relative power consumption of the benchmark functions measured in the above experiment, the PRE was calculated and presented as a bar graph. Additionally, the RS value was calculated by applying the derived data listed in Table 5 to Equation (2) and presents the line in the subplots. The $Slope(g)$, $Intercept(g)$, and $R(g)$ used here employed the result of the regression analysis in Figure 8. This is acceptable because the experiment was conducted to evaluate the proposed model in the same environment as the code profiling environment, which should be conducted in the pre-stage for regression analysis. The primary purpose of this work was to correctly predict the PRE order in a program. In this regard, the results showed that RS demonstrates a similar trend to PRE. In other words, the higher the bar that represents the PRE value, the higher the measured RS value, thereby signifying that function relocation is efficient. In other words, RS achieves exactly the same order as PRE. Consequently, we verified that the proposed model accurately predicts the functions relocated to SRAM in function-complicated programs in the order of reduced power.

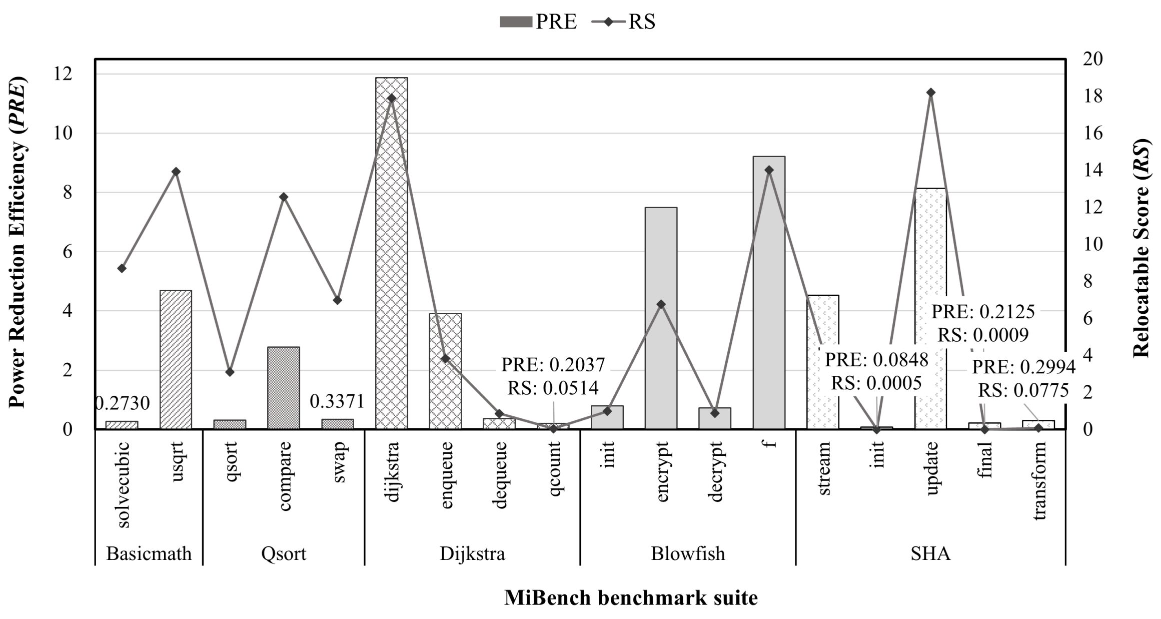


Figure 10. Power Reduction Efficiency (PRE) value and Relocatable Score (RS) value correlation.

7. Conclusions

The design of low-power embedded systems requires consideration of the limitations of hardware components. Previous studies identified the potential for power reduction using a code migration methodology based on function relocation on the software level, without modifying critical system components such as the hardware. In addition, other researchers uniformly reasoned that the location of the code is crucial when considering the heterogeneous memory characteristics. This means that a function existing in volatile memory consumes less power compared to a function in non-volatile memory.

In this work, code profiling, which enables us to understand the actual program flow, was preferentially performed to confirm the effect of function relocation based on the instruction groups. This step was necessary to provide convincing evidence that the efficiency of power reduction due to relocation could be attributed to the instruction groups. Thus, the assembly codes related to each group were inserted and the power consumption for each insertion was measured in an actual system. The profiling result indicated an increase in the power reduction efficiency due to function relocation in the following order: The highest was achieved with the Branch and Control group and the lowest with the Register Load and Store group. Furthermore, a prior relocation-scoring model, which guesses the effective relocation order in a program, was proposed. Based on the profiling result, the model calculates the Relocatable Score (RS) and compares it to the Power Reduction Efficiency (PRE). Thus, appropriate benchmarks were selected as case studies. As a consequence, the results showed a consistent trend between the scores and the efficiency. In other words, a function allocated a high RS value by the model ensures strong power reduction compared to the other functions in the benchmark program. A clear criterion for selecting a target function for relocation is presented. Moreover, the result is of great significance because it indicates the possibility to predict the effect of the order of function-level code relocation by using only the ratio of instruction groups.

With reference to one program, the result is considerably significant. However, it is not possible to score multiple programs on one basis. That is the only limitation of the proposed model, namely that the effect of relocation cannot be predicted only by the proposed scored value. Therefore, it would be necessary to additionally consider instruction-specific characteristics and combinations between instructions, rather than simply classifying instructions into groups of instructions.

Author Contributions: H.C. and Y.K.: Conceived and designed the methodology and the model and performed the experiments and analyzed the data; S.P.: Conceptualization, validation and review; All authors worked on this manuscript together, and all authors have read and approved the final manuscript.

Funding: This research was funded by the National Research Foundation of Korea funded by the Korean Government (NRF-2017R1D1A1B03030393).

Acknowledgments: This research is an extension of the following paper: Hayeon Choi, Youngkyoung Koo and Sangsoo Park, "Extensive Instruction Code Profiling for Effective Function-Level Code Relocation," International Symposium on Computer, Consumer and Control (IS3C), Taichung, Taiwan, 2018, pp. 38-41. This research was supported by the National Research Foundation of Korea funded by the Korean Government (NRF-2017R1D1A1B03030393). Sangsoo Park is the corresponding author.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Hua, S.; Qu, G. Approaching the maximum energy saving on embedded systems with multiple voltages. In Proceedings of the IEEE/ACM International Conference on Computer-Aided Design, San Jose, CA, USA, 9–13 November 2003; p. 26.
2. Quan, G.; Hu, X. Energy efficient fixedpriority scheduling for real-time systems on variable voltage processors. In Proceedings of the Design Automation Conference, Las Vegas, NV, USA, 18–22 June 2001; pp. 828–833.
3. Yang, L.; Dick, R.P.; Lekatsas, H.; Chakradhar, S. Online memory compression for embedded systems. *ACM Trans. Embed. Comput. Syst.* **2010**, *9*, 27:1–27:30. [CrossRef]
4. Steinke, S.; Grunwald, N.; Wehmeyer, L.; Banakar, R.; Balakrishnan, M.; Marwedel, P. Reducing energy consumption by dynamic copying of instructions onto onchip memory. In Proceedings of the 15th IEEE International Symposium on System Synthesis, Kyoto, Japan, 2–4 October 2002; pp. 213–218.
5. Choi, H.; Koo, Y.; Park, S. Quantitative analysis of power consumption for low power embedded system by types of memory in program execution. *J. Korea Multimedia Soc.* **2016**, *17*, 1179–1187. [CrossRef]
6. Choi, H.; Koo, Y.; Kim, N.; Kim, J.; Jung, B. A study on the low power memory for efficient embedded system. *J. WISET Junior Sci. Technol. Res. Rep.* **2016**, *6*, 275–281.
7. Choi, H.; Koo, Y.; Park, S. Segment-aware energy-efficient management of heterogeneous memory system for ultra-low-power IoT devices. In Proceedings of the 2nd International Multidisciplinary Conference on Computer and Energy Science, Split, Croatia, 12–14 July 2017; pp. 1–6.
8. Choi, H.; Koo, Y.; Park, S. A novel function complexity-based code migration policy for reducing power consumption. *J. Commun. Softw. Syst.* **2018**, *1*, 68–74. [CrossRef]
9. Eggenberger, M.; Radetzki, M. Optimal memory selection for low power embedded systems. In Proceedings of the 12th International Workshop on Intelligent Solutions in Embedded System, Ancona, Italy, 29–30 October 2015; pp. 11–16.
10. MSP432E401Y. Available online: http://www.ti.com/product/MSP432E401Y (accessed on 8 June 2018).
11. MSP432P401R. Available online: http://www.ti.com/product/MSP432P401R (accessed on 9 April 2017).
12. ULINKpro Debug and Trace Unit. Available online: http://www2.keil.com/mdk5/ulink/ulinkpro (accessed on 22 June 2018).

13. XDS110 Debug Probe User's Guide. Available online: http://www.ti.com/lit/ug/sprui94/sprui94.pdf (accessed on 17 May 2017).
14. The ARM C Compiler. Available online: https://www.cl.cam.ac.uk/teaching/1998/CompDesn/fromlecturer/htmlman/armcc.html (accessed on 29 October 2017).
15. TI Compiler Information. Available online: http://processors.wiki.ti.com/index.php/TI_Compiler_Information (accessed on 15 August 2018).
16. ARM and Thumb Instruction Set Overview. Available online: http://www.keil.com/support/man/docs/armasm/armasm_dom1359731139853.htm (accessed on 3 July 2018).
17. MDK Microcontroller Development Kit. Available online: http://www2.keil.com/mdk5/ (accessed on 2 December 2017).
18. Code Composer Studio (CCS) Integrated Development Environment (IDE). Available online: http://www.ti.com/tool/CCSTUDIO (accessed on 29 March 2017).
19. Non-Intrusive Debugging with ETM Trace. Available online: https://www.iar.com/support/resources/articles/non-intrusive-debugging-with-etm-trace/ (accessed on 3 September 2017).
20. EnergyTrace for MSP432. Available online: http://processors.wiki.ti.com/index.php/EnergyTraceforMSP432 (accessed on 11 March 2017).
21. Chatterjee, S.; Hadi, A.S. *Regression Analysis by Example*, 5th ed.; Wiley Series in Probability and Statistics: Chichester, UK, 2015; p. 424, ISBN 978-111-912-273-9.
22. Guthaus, M.R.; Ringenberg, J.S.; Ernst, D.; Austin, T.M.; Mudge, T.; Browni, R.B. MiBench: A free, commercially representative embedded benchmark suite. In Proceedings of the Fourth Annual IEEE International Workshop on Workload Characterization, Austin, TX, USA, 2 December 2001; pp. 3–14.