An Efficient Function Inlining Scheme for Resource-Constrained Embedded Systems*

BONGJAE KIM1, YOOKUN CHO1 AND JIMAN HONG2
1School of Computer Science and Engineering
Seoul National University
Seoul, 151-172 Korea
E-mail: {bjkim, ykcho}@os.snu.ac.kr
2School of Computing
Soongsil University
Seoul, 156-743 Korea
E-mail: jim@ssu.ac.kr

Function inlining schemes are used to reduce the execution time of a program, but function inlining usually increases the code space due to the use of inlined functions. There are extreme limitations of memory space and battery capacity in embedded systems such as wireless sensor nodes. Function inlining should be performed selectively. If we reduce the execution time of a program by means of function inlining, we can reduce the amount of energy that is consumed. Therefore, function inlining is useful in resource-constrained embedded systems. Basic function inlining schemes were proposed in the previous studies. However, these schemes were too coarse-grained, and did not analyze the impact of function inlining on resource-constrained embedded systems. In this paper, we propose a fine-grained function inlining scheme which considers the execution rate of the individual functions. We also present an analysis of the impact of function inlining schemes on resource-constrained embedded systems, in terms of energy consumption and code memory usage. Based on experimental results, we show that the proposed fine-grained function inlining scheme can improve the level of energy efficiency.

Keywords: function inlining, embedded systems, wireless sensor networks, energy efficiency, memory efficiency

1. INTRODUCTION

Presently, the computer system paradigm is shifted to embedded systems that are dedicated to specific tasks. For example, wireless sensor networks consist of hundreds or thousands of wireless sensor nodes deployed in remote regions to sense events that occur in the environment. Typically, sensor nodes have limited resources. For example, a sensor node may have 2~10 KB RAM and 60~128 KB of flash memory as well as a limited battery capacity [1-3, 14]. Therefore, it is necessary to consider the efficiency of these systems in terms of both memory space and energy consumption. A number of optimization techniques for embedded systems have been developed to improve the efficiency. Examples are techniques for program code, register allocation, scheduling, and memory
In this paper, we propose a source-level code optimization scheme known as function inlining. Function inlining is a well-known technique that is used in many programming languages [4]. Function inlining eliminates the time overhead when a function is called, such as that needed for parameter passing, call and return instructions, instruction pipeline stalls, and saving and restoring register contents. In function-oriented programming languages such as C language, function inlining schemes have been used to reduce the execution time while unfortunately increasing the code size in the program image. We can reduce the energy consumption by reducing the execution time. In general-purpose computing systems such as personal computers, stable storage devices such as a HDD do not generally have resource constraints in terms of code memory space or battery capacity. Therefore, function inlining is highly applicable to these resource-abundant systems [4, 5].

In embedded systems such as a wireless sensor node, the upper bound of the size of the program code is the size of its embedded flash memory, and the energy efficiency is related to the lifetime of these devices. With function inlining, it is necessary to reduce energy consumption while utilizing the residual code memory space in embedded systems. In these resource-constrained embedded systems, selective function inlining is very important for the efficient use of the residual code memory space and to maximize energy efficiency [6]. In these embedded systems, code memory space is severely limited. Therefore, in order to avoid an explosion in the code size, only a limited set of functions should be candidates for function inlining.

Several studies have sought to use function inlining in embedded systems with code memory constraints [6-9]. These schemes calculate the expected benefits of function inlining based on the expected increase in code memory and decrease in the execution time for each function. They then selectively perform function inlining according to the calculated benefits. However, the existing schemes are coarse-grained, as they do not consider the execution rate of each function call. This can degrade the performance of function inlining [6, 7]. In addition, existing works do not analyze the impact of function inlining on real-world embedded systems in terms of both energy consumption and code memory usage.

In this paper, we present a fine-grained function inlining scheme. We also analyze the impact of function inlining schemes on resource-constrained embedded systems in terms of both energy consumption and code memory usage. Based on the experimental results, we show that the proposed fine-grained function inlining scheme can significantly improve both energy efficiency and memory utilization.

The rest of this paper is organized as follows. In section 2, we present existing works related to function inlining. Section 3 presents the requirements of resource-constrained embedded systems and the system model used in this paper. Section 4 presents the design for the fine-grained function inlining scheme and its implementation. In section 5, we evaluate the impact and performance of function inlining schemes on resource-constrained embedded systems through experimental results in terms of both energy consumption and memory space usage. Finally, the conclusion and future work are discussed in section 6.
2. RELATED WORKS

In this section, we briefly present previous works related to function inlining. We determine the features of each scheme, and analyze the pros and cons one by one.

2.1 Branch-And-Bound Algorithm

One earlier study proposed a novel function inlining scheme that considered the demand of embedded systems [6]. The main purpose of this algorithm is to reduce the program execution time by minimizing the number of dynamic function calls. It selects the candidate functions to be inlined based on the Branch-and-Bound (B&B) algorithm. It generates an inline vector and calculates the total code size recursively up to the leaf functions, which do not contain any function calls.

Fig. 1 shows an example of the inline vector used in the B&B algorithm. In the inline vector, each bit of the inline vector denotes a corresponding function of a program code. Therefore, the inline vector presents the inlining status of individual functions. The B&B algorithm finds the minimum number of dynamic function calls when the total size of codes is less than the size of the memory. It is simple to design and implement, but the complexity of this algorithm is \( O(2^n) \), where \( n \) denotes the number of functions of a program. A brute-force exhaustive search is clearly required. This search is very time consuming.

![Inline Vector Example](image)

Fig. 1. An example of the inline vector.

2.2 Heuristic Algorithm

In another study [7], a heuristic-based function inlining scheme was proposed to find candidate functions for function inlining. In that paper, they define a heuristic function inlining variable as follows,

\[
rebate \text{ ratio} = \frac{\text{function calling frequency}}{\text{increased code size}}.
\] (1)

In this equation, the numerator refers to how often this function is called by others, and the denominator signifies the increased size of the code as caused by the inlined function. Inlining a function with a high rebate ratio will lead to better performance compared to the use of a low rebate ratio. To model the determining steps of inlining candidate functions, they use a graph algorithm that is represented as \( G = (V, E) \); nodes are functions and edges are the weight based on the rebate ratio. Fig. 2 shows an example of a graph that is shifted by function inlining.
Before Function Inlining

\[ f_3 \text{ is inlined to } f_1 \]

\[ f_0 \quad \w_1 \quad \w_2 \quad \w_3 \quad \w_4 \]

After Function Inlining

\[ f_3' \quad \w_1' \quad \w_2' \quad \w_3' \quad \w_4' \]

\[ f_1' \quad \w_1' \quad \w_2' \quad \w_3' \quad \w_4' \]

**Fig. 2. The graph-shift by function inlining.**

In Fig. 2, nodes represent functions and edges which have their own weighting value as denoted as \( w \). Each \( w \) refers to the function calling frequency between two nodes or functions. In Fig. 2, we assume that the function \( f_3 \) has the highest rebate ratio among all functions. Therefore, the function \( f_3' \) is inlined to the function \( f_1 \), and \( f_1 \) becomes \( f_1' \) due to the inlined function \( f_3 \). This heuristic function inlining scheme provides a simple and efficient mechanism to find the inlining candidate functions. However, this heuristic scheme does not consider code sequences such as loops and recursive statements. Furthermore, the cost of graph management is too high.

### 2.3 Aggressive Function Inlining

Other research considered that an I-cache miss would arise due to function inlining when function inlining was used to optimize the performance [8]. To solve the I-cache miss problem, global code reordering was used. The I-cache miss problem can be reduced by global code reordering. The global code reordering scheme with function inlining guarantees the separation between cold and hot segments of the inlining functions. In their study, they distinguished the cold and hot segments using the information gathered from a representative workload. They implemented this scheme based on edge profiling. This scheme is termed aggressive function inlining because it does not consider the boundary of the increased code size. It simply adds the hot or cold segment information to the call graph and performs global code reordering with function inlining. This function inlining scheme can optimize and improve the system performance effectively. However, it requires a compiler and hardware-level support. Moreover, it is designed to run on a general-purpose system.

### 2.4 Partial Inlining with Function Outlining

Another study suggested partial inlining scheme with function outlining [9]. Function outlining makes the original function smaller. In general, a large function consists of two parts. The first part is a hot region that is executed much more frequently, while the second part is a cold region that is executed much less frequently. During the function inlining operation, problems are caused by including cold code segments in hot functions. When applying partial inlining, a compiler outlines the cold segments in a hot function.
After outlining these cold segments, the hot function becomes smaller and thus can be more easily inlined. Based on this idea, a framework was suggested that uses a WHIRL tree. With partial inlining coupled with function outlining, the system performance can be improved.

3. REQUIREMENTS AND SYSTEM MODEL

In this section, we present our considerations and essential requirements for resource-constrained embedded systems. We then explain the system model used in this paper to describe function inlining scheme.

3.1 Requirements

In resource-constrained embedded systems such as wireless sensor networks, each sensor node has limited battery power, a low processing power, and a small memory space. Therefore, a new function inlining scheme running on a resource-constrained embedded system must consider the requirements below. Fig. 3 shows the requirements of embedded systems.

- **Energy usage**: most embedded systems operate under the limited power of a battery. Therefore, energy efficiency is directly related to the lifetime of the system. Efficient energy consumption is the most important factor when considering the design and implementation of a new function inlining scheme.
- **Execution time**: the amount of CPU using time is equal to the amount of energy consumption. This is a critical issue related to the reduction in the operation cost or the execution time through the use of simple architectures and optimization techniques. In short, if we reduce the execution time, we can reduce the energy consumption.
- **Minimize additional memory demand**: though a new function inlining scheme may be efficient in terms of its execution time, it may not be applicable to an embedded systems such as a sensor node if it requires too much memory space to manage or operate its algorithm. Therefore, the code memory space should be minimized and implemented in an optimal manner.

![Fig. 3. Requirements of embedded systems.](image-url)
We designed and implemented our fine-grained function inlining scheme considering these requirements.

3.2 System Model

Table 1 shows some notations and descriptions used in this paper to present function inlining schemes.

<table>
<thead>
<tr>
<th>Notation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( f_i )</td>
<td>( i )th function ( f_i )</td>
</tr>
<tr>
<td>( c_j(f_i) )</td>
<td>( j )th function call instruction which calls ( f_i )</td>
</tr>
<tr>
<td>( n_c(f_i) )</td>
<td>Number of function call instructions which calls ( f_i )</td>
</tr>
<tr>
<td>( s_c(f_i) )</td>
<td>Size of codes of ( f_i )</td>
</tr>
<tr>
<td>( t_c(f_i) )</td>
<td>Time overhead when calling ( f_i )</td>
</tr>
<tr>
<td>( \lambda_c(c_j(f_i)) )</td>
<td>Execution rate of a call instruction ( c_j(f_i) )</td>
</tr>
<tr>
<td>( D_c(f_i) )</td>
<td>Discriminant when using coarse-grained function inlining scheme for a function ( f_i )</td>
</tr>
<tr>
<td>( D_f(f_i, c_j(f_i)) )</td>
<td>Discriminant when using fine-grained function inlining scheme for a call instruction ( c_j(f_i) )</td>
</tr>
</tbody>
</table>

In Table 1, \( f_i \) is the \( i \)th function in a program code and \( c_j(f_i) \) is the \( j \)th function call instruction which calls the function \( f_i \). In addition, \( n_c(f_i) \) is the number of function call instructions that calls the function \( f_i \); \( s_c(f_i) \) is the code size of the function \( f_i \); and \( t_c(f_i) \) is the time overhead when calling the function \( f_i \). The notation \( \lambda_c(c_j(f_i)) \) refers to the execution rate of the function call instruction \( c_j(f_i) \), and the \( D_c(f_i) \) and \( D_f(f_i, c_j(f_i)) \) notations are discriminant values that are used when using a coarse-grained and a fine-grained function inlining scheme for the function \( f_i \) respectively. With these notations, we describe the coarse-grained and the fine-grained function inlining scheme in detail.

4. FUNCTION INLINING FOR RESOURCE-CONSTRAINED EMBEDDED SYSTEMS

This section presents function inlining schemes for resource-constrained embedded systems. We then present the design and implementation our fine-grained function inlining scheme.

4.1 Function Inlining on Embedded Systems

We can reduce the energy consumption through the use of function inlining. In function-oriented programming languages, function inlining removes function calling overhead operations such as parameter passing, call and return instructions, instruction pipeline stalls, and the saving and restoring of register contents. Therefore, if we use function inlining, the execution time is reduced. Function inlining can be used to improve
energy efficiency.

Though the function inlining scheme reduces both energy consumption and the execution time, it increases the size of the code. Thus, the code memory space usage will increase. General-purpose computing systems such as a PC are not severely constrained in terms of their program storage or code memory space. Thus, we can perform function inlining aggressively in order to improve the performance.

However, in the case of embedded systems, flash memory substitutes for the stable storage form of a HDD. Moreover, flash memory is typically much smaller than a HDD. Typically, embedded systems have several resource constraints. The most important factor is their limited battery power. For example, some sensor nodes can harvest energy using a solar cell or a small windmill, but most do not have these facilities. Therefore, the lifetime of a sensor node is limited by its battery capacity. For this reason, energy efficiency is directly related to its lifetime. In these embedded systems, function inlining can improve energy efficiency by reducing the execution time, thus allowing the lifetime to be maximized. In addition, embedded systems have severely limited memory space. Because the size of the memory is not sufficient, function inlining should be performed selectively and not aggressively.

To meet the above constraints of embedded systems, function inlining should consider its effect on the code size overhead and the reduction of the execution time.

### 4.1.1 Function inlining schemes

Previous works proposed function inlining schemes based on a discriminant value \[6, 7\]. This discriminant value considers the code size increment and the amount of reducible execution time for each function. In these function inlining schemes, the discriminant value is calculated as follows,

\[
D_c(f_i) = \frac{\sum_{j=1}^{n_i(f_i)} t_i(f_i) \times \lambda_j(c_j(f_i))}{n_i(f_i) \times s_i(f_i)}. \quad (2)
\]

For the discriminant value \(D_c(f_i)\), the numerator is the amount of reducible execution time when a function \(f_i\) is inlined. The denominator is the increased code size caused by function inlining. Therefore, a larger \(D_c(f_i)\) value means that the function \(f_i\) is better for maximizing both the energy efficiency and the utilization of the memory space.

When performing function inlining, previous works simply sort all \(D_c(f_i)\) values of function calls in a descending order and then perform function inlining until the residual code memory space is not sufficient for inlining. In these coarse-grained function inlining schemes, the basic idea is very simple. This leads to good performance compared to non-inlining schemes. However, if the calling rate of function \(f_i\) is significantly different between each function call instruction, the coarse-grained function inlining scheme cannot guarantee energy efficiency or code memory space efficiency because the effect of the function inlining of each function call instruction is significantly different. This makes it necessity to consider calling rate of each function call instruction.
Fig. 4 shows an example of the C program code. In this example C program, the function \( f_{\text{bar}} \) is called multiple times in a for loop and in the main body of the program. In this example, although \( n(c) \) is 101, \( \lambda(c_j(f_{\text{bar}})) \) is significantly different in each case. For example, \( \lambda(c_1(f_{\text{bar}})) \) is 100 times greater than \( \lambda(c_2(f_{\text{bar}})) \), \( \lambda(c_3(f_{\text{bar}})) \), …, and \( \lambda(c_{101}(f_{\text{bar}})) \). Therefore, to inline the \( c_1(f_{\text{bar}}) \) function is reasonable as the first priority because the effect of the function inlining of \( c_1(f_{\text{bar}}) \) is 100 times greater than the that of the others. If we simply use the coarse-grained function inlining scheme, every \( f_{\text{bar}}() \) call instructions will be inlined at once. This requires excessive space overhead compared to inlining only an instruction \( c_1(f_{\text{bar}}) \) at once. Therefore, this course-grained function inlining scheme is not suitable for resource-constrained embedded systems.

Hence, we propose a fine-grained function inlining scheme for resource-constrained embedded systems. The calling rate of each function call instruction is considered in our fine-grained function inlining scheme. The fine-grained inlining scheme performs function inlining for each function call instruction \( c_j(f_i) \). Thus, the discriminant value is calculated and maintained for each function call instruction \( c_j(f_i) \). The discriminant value is calculated as follows,

\[
D_{f_i}(f_i, c_j(f_i)) = \frac{\sum_{i=1}^{n_c(f_i)} \lambda(c_j(f_i)) S_i(f_i)}{s_i(f_i)}.
\]  

(3)

For the discriminant value \( D_{f_i}(f_i, c_j(f_i)) \), the denominator does not have \( n_c(f_i) \) because the fine-grained function inlining scheme performs function inlining for each call instruction \( c_j(f_i) \). Therefore, our fine-grained function inlining scheme can maximize the residual memory space utilization.

Algorithm 1 shows the fine-grained function inlining scheme in detail.

\begin{algorithm}
\begin{algorithmic}
\STATE \textbf{Note:} This algorithm is performed during the compile-time of the target image.
\STATE \textbf{CALCULATE} \( D_{f_i}(f_i, c_j(f_i)) \) for all call instruction \( c_j(f_i) \)
\STATE \textbf{SORT} all \( D_{f_i}(f_i, c_j(f_i)) \) in descending order
\FORALL {\( D_{f_i}(f_i, c_j(f_i)) \)}
\IF {residual code memory space \( \geq S_i(f_i) \)} \THEN
\STATE \textbf{PERFORM} function inlining for a call instruction \( c_j(f_i) \)
\ELSE
\ENDIF
\ENDFOR
\end{algorithmic}
\end{algorithm}
The function inlining is available on the source-code level. Algorithm 1 is performed during the compile time of the target image. First, in the fine-grained function inlining scheme, $D(f_i, c(f_i))$ is calculated for all call instruction $c(f_i)$. After that, all $D(f_i, c(f_i))$ are sorted in a descending order. Then, function inlining is performed for all $c(f_i)$. If the amount of residual code memory space is not sufficient for inlining, function inlining is terminated. The actual implementation is explained in the following subsection.

### 4.1.2 Implementation

To implement the fine-grained function inlining scheme on an actual embedded system, we need to know the parameters presented in Table 1. The following are tools for the fine-grained function inlining scheme: the Function Size Checker (FSC), Function Call Counter (FCC), Calling Rate Analyzer (CRA), and Function Inliner (FI). The FSC checks the size of a function. The FCC counts the total number of call instructions of a function. The CRA counts the calling rate of a function. Finally, the FI performs function inlining. We implemented the FSC and FCC.

Fig. 5 shows the entire building process when using the fine-grained function inlining scheme. The size of each function and the number of call instructions are analyzed from the assembly code by the C source code, and the calling rates and the time overhead are measured by actually running the target image. With the above information, the Function Inliner modifies the original C codes. Finally, the modified C code is compiled into the final target image. In this paper, we manually measured the calling rate and time overhead for each function.

### 5. PERFORMANCE EVALUATION

In this section, we evaluate performance and analyze the impact of function inlining
schemes on resource-constrained embedded systems in terms of both energy consumption and code memory space usage.

5.1 Experimental Setup

In our experiment, we used Octacomm’s Nano-24 wireless sensor platform [10], which is very similar to the UC Berkeley’s MICAZ mote. Table 2 shows the specifications of the Nano-24 platform. We used Nano-Qplus ver. 2.3.0 [11-13] as a base operating system as well as their existing sensor applications. Nano-Qplus is flexible, dynamic, and easy manageable for sensor network application programmers.

<table>
<thead>
<tr>
<th>Component</th>
<th>Model</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>ATMega128L</td>
<td>Low-power 8 bit microprocessor</td>
</tr>
<tr>
<td></td>
<td>Flash</td>
<td>128 kB</td>
</tr>
<tr>
<td>Memory</td>
<td>SRAM</td>
<td>4 kB</td>
</tr>
<tr>
<td></td>
<td>EEPROM</td>
<td>4 kB</td>
</tr>
<tr>
<td>R/F</td>
<td>CC2420</td>
<td>2.5 GHz channel</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Zigbee support</td>
</tr>
</tbody>
</table>

In our experiments, we compared the following three types of function inlining schemes:

- Non-Inlining Scheme (NIS)
- Coarse-grained Inlining Scheme (CIS)
- Fine-grained Inlining Scheme (FIS)

In these schemes, CIS is identical to the existing function inlining schemes [4, 5]. NIS does not perform any function inlining. FIS is the proposed function inlining scheme. We used two examples of sensor applications to evaluate and compare the performance of the three schemes.

Fig. 6 shows overall structure of a light sensing application. In this application, a TX node measures the current lightness of the room and transmits the sensed lightness data to a RX node via a wireless communication channel. The RX node then receives and forwards the data packet to the data collector via a serial line. Between these communications, each node turns on and off their LEDs.

The second application is a LED control program which toggles the LEDs on the sensor node. It is similar to the example program described in Fig. 4. Using this example, we can compare the space and time efficiency of FIS and CIS. In Fig. 7, the execution rate of two function calls in a for loop is 50 times greater than that of the other functions. On the outside of the for loop, func_LED_ON() and func_LED_OFF() are called 50 times each.
5.2 Experimental Results and Evaluation

We evaluated the performance of the three inlining schemes in terms of both the execution time and the total code size. First, the execution time denotes the amount of reducible CPU operation; it is directly related to the energy consumption of the sensor nodes. Secondly, the total size of the code denotes the overhead of the code when using a function inlining scheme.

5.2.1 Lightness sensing application

Fig. 8 shows the one-round execution time of the TX and RX nodes in the lightness sensing application. Based on Fig. 8, the execution times of CIS and FIS are slightly reduced compared to that of NIS. The execution time of the TX node of NIS is about 39 ms. The execution time of the TX node of FIS and CIS are about 36 ms. In the case of the RX node, the execution time of NIS is about 12 ms. The execution time of FIS and CIS are about 9 ms.

Fig. 9 shows the total code size of the lightness sensing application. In this result, CIS and FIS have less code size overhead compared to NIS. In the case of the RX node, the total code size of NIS is 47591 bytes. The total code size of CIS and FIS are 48471 bytes and 49424 bytes, respectively. The increase in the code size of FIS is about 4%. Based on Figs. 8 and 9, we note that CIS and FIS show slightly better execution times while using more code memory space. However, we cannot clearly distinguish the performance of FIS compared to CIS. Therefore, we performed the second experiments on the LED application.
5.2.2 LED application

To compare the performance of CIS and FIS, we evaluated the performances during a LED application. We also considered the upper bound of the code memory space to utilize the code space fully while minimizing the execution time. In other words, we restricted the use of the code memory space for function inlining.

Fig. 10 shows the execution time of the LED application. The execution time of NIS, CIS, and FIS are about 2.4 ms, 1.34 ms, and 0.91 ms, respectively. In the case of CIS, the execution time is approximately half of that of NIS, and FIS shows better performance than CIS. We noted that FIS reduces the execution time to 62.5% that of NIS. Therefore, energy efficiency is improved when using the proposed scheme.

Fig. 11 shows the total code size of the simple LED application. In this result, CIS uses 585 bytes to inline functions, whereas FIS uses 712 bytes. In the case of FIS, the increase in the code size is about 3%. Based on these results, we note that the proposed scheme FIS shows better performance in terms of the execution time compared with CIS.

5.3 Static Analysis of Reduced Energy Consumption

To analyze the reduction in energy consumption by function inlining, we performed
static analysis of reduced energy consumption by analyzing the power consumption of the CPU. To calculate the power consumption, it is necessary to know the operating voltage and current. The ATmega128L is the CPU used in our experiment. The ATmega128L supports 4MHz and 8MHz operating frequencies. This allows the selection of the operating frequency. When the operating frequency is 4MHz, the operating voltage is 3V. At an operating frequency of 4MHz, the inputted current of the IDLE state and the ACTIVE state are 2mA and 5mA, respectively. Similarly, when the operating frequency is 8MHz, the operating voltage is 5V. At an operating frequency of 8MHz, the inputted current of the IDLE state and the ACTIVE state are 8mA and 17mA, respectively [1].

With the above information, we can analyze the reduced energy consumption using the equation below. $E_{\text{REDUCED}}$ denotes the reduced energy consumption. $I_{\text{ACTIVE}}$ denotes the current of the ACTIVE state of the ATmega128L. $I_{\text{IDLE}}$ denotes the current of the IDLE state of the ATmega128L. $V_{\text{OP}}$ denotes the operating voltage. Finally, $T_{\text{REDUCED}}$ denotes the reduced execution time due to function inlining

$$E_{\text{REDUCED}} = (I_{\text{ACTIVE}} - I_{\text{IDLE}}) \times V_{\text{OP}} \times T_{\text{REDUCED}}.$$  (4)

Fig. 12 shows the reduced energy consumption according to the reduced execution time using the ATmega128L. The reduction in the energy consumption increases with a greater reduction in the execution time. When the operating frequency is 8MHz, the reduction in the energy consumption is greater than 4MHz.
Fig. 12. Analysis of reduced energy consumption.

6. CONCLUSION AND FUTURE WORK

In resource-constrained embedded systems, memory space and battery capacity are highly constrained compared to general-purpose systems such as a PC. Therefore, function inlining should be performed selectively while maximizing the residual memory space utilization and minimizing energy consumption. In earlier works, basic function inlining schemes were proposed and adapted to code memory-limited systems. However, those schemes were too coarse-grained, as they did not consider the execution rate of each function call instruction. In this paper, we proposed a fine-grained function inlining scheme. In our fine-grained function inlining scheme, we considered the calling rate of each function call instruction. Our experimental results showed that the proposed fine-grained function inlining scheme significantly improves energy efficiency.

We have plans to extend our work to combine the proposed scheme with a function outlining scheme. In addition, we will study data and stack space efficiency when using the function inlining and outlining schemes to improve the efficiency of the resource-constrained embedded systems.

REFERENCES


Bongjae Kim received his B.S. degree in Computer Engineering from Kwangwoon University, Seoul, Korea, in 2008. He has been with School of Computer Science and Engineering, Seoul National University since 2009, where currently he is a Ph.D. candidate student. His research interests include operating systems, fault tolerance computing systems, embedded systems, and sensor networks.

Yookun Cho received the B.E. degree from Seoul National University, Korea, in 1971 and the Ph.D. degree in Computer Science from the University of Minnesota at Minneapolis in 1978. He has been with the School of Computer Science and Engineering, Seoul National University since 1979, where he is currently a Professor. He was a visiting assistant professor at the University of Minnesota during 1985 and a director of the Educational and Research Computing Center at Seoul National University from 1993 to 1995. He was president of the Korea Information Science Society during 2001. He was a member of the program committee
of the IPPS/SPDP in 1997 and the International Conference on High Performance Computing from 1995 to 1997. His research interests include operating systems, algorithms, system security, and fault-tolerant computing systems. He is a member of the IEEE.

Jiman Hong received the B.S. degree in Computer Science from Korea University, Seoul Korea in 1994 and the M.E. and Ph.D. degrees in Computer Engineering from Seoul National University, Seoul Korea, in 1997, and 2003, respectively. He was an assistant professor at Kwangwoon University from 2004 to 2006. He has been with School of Computing, Soongsil University, Seoul, Korea, since 2007, where currently he is a professor. From 2000 to 2003, he served as a Chief of Technical Officer in the R&D center of GmanTech Incorporated Company, Seoul, Korea. His research interests include embedded operating system, fault tolerance computing systems, distributed computing systems, and sensor network systems.