Implementation and Evaluation of an On-Demand Parameter-Passing Strategy for Reducing Energy*

M. Kandemir
CSE Department
Penn State University
University Park, PA, 16802

I. Kolcu
Computation Department
UMIST
Manchester M60 1QD, UK

W.Zhang
CSE Department
Penn State University
University Park, PA, 16802

Abstract

In this paper, we present an energy-aware parameter-passing strategy called on-demand parameter-passing. The objective of this strategy is to eliminate redundant actual parameter evaluations if the corresponding formal parameter in a subroutine is not used during execution. This on-demand parameter-passing is expected to be very successful in reducing energy consumption of large, multi-routine embedded applications at the expense of a slight implementation complexity.

1. Introduction

It is possible to attack the energy consumption problem from different angles. Throughout the years, circuit designers developed numerous techniques to reduce energy consumption (see [2] and the references therein). Recent years have also witnessed several architectural and system level optimizations. Among these are energy-efficient management of hardware components and use of low-power operating modes. On the software side, operating system (OS) based studies targeted at reducing energy consumption through runtime monitoring, scheduling, and voltage/frequency scaling (e.g., [11]). More recently, compiler researchers (e.g., [10]) proposed several compilation techniques for optimizing energy behavior of applications without impacting their performance severely. It is also possible to reduce energy consumption using algorithmic/application level optimizations [3].

We strongly believe that addressing ever increasing energy consumption problem of integrated circuits must span multiple areas. While advances in circuit, architecture, OS, application, and compiler areas are promising, it might also be important to consider programming language support for low power. This issue is very critical because programming language defines the interface between algorithm/application and the underlying architecture/execution environment. The types of optimizations that can be performed by the compiler and possible architectural hooks that can be exploited by the runtime system are also determined and/or controlled by the programming language. Unfortunately, to the best of our knowledge, there has not been a study so far for evaluating different language features from an energy consumption perspective.

Parameter-passing mechanisms are the ways in which parameters are transmitted to and/or from called subprograms [9]. Typically, each programming language supports a limited set of parameter-passing mechanisms. In C, one of the most popular languages in programming embedded systems, all parameter evaluations are done before the called subprogram starts to execute. This early parameter evaluation (i.e., early binding of formal parameters to actual parameters), while it is preferable from the ease of implementation viewpoint, can lead to redundant computation if the parameter in question is not used within the called subprogram.

In this paper, we present an energy-aware parameter-passing mechanism that tries to eliminate this redundant computation when it is detected. The proposed mechanism, called *on-demand parameter-passing,* computes value of an actual parameter if and only if the corresponding formal parameter is actually used in the subroutine. It achieves this by using compiler's help to postpone the computation of the value of the actual parameter to the point (in the subroutine code) where the corresponding formal parameter is actually used. It should be emphasized that our objective is not just to eliminate the computation of the value of the actual parameter but also all other computations that lead to the computation of that parameter value (if such computations are not used for anything else). Our on-demand parameter-passing mechanism is entirely transparent to the user and does not modify the meaning of the application irrespective of whether there exists redundant computation due to parameter-passing or not. Our work is complementary to the studies in [4, 5, 12]. There is also work on hardware-based common case exploitation (e.g., [8]). In contrast, our approach is software oriented and targets unused formal parameters rather than common case computations.

2. Review of Parameter Passing Mechanisms

Subroutines are the major programming language structures for enabling control and data abstraction [9]. The in-

*This work was supported in part by NSF CAREER Award #0093082.

terface of the subroutine to the rest of the application code is indicated in its *header* using subroutine name and its parameters. The parameters listed in the subprogram header are called *formal parameters*. Subprogram call statements must include the name of the subroutine and a list of parameters, called *actual parameters*, to be bound to the formal parameters in the subprogram header. In a typical implementation, at the subroutine invocation time, the actual parameters are computed and passed (copied) to formal parameters using the *parameter access path*. After the execution of the subroutine, depending on the *parameter return path* used, the values of the formal parameters are copied back to actual parameters. The parameter access path and parameter return path implementations depend on the parameter-passing mechanism adopted and discussed below for the C language.

In C, each subroutine parameter is passed using one of two mechanisms: *call-by-value* (CBV) and *call-by-reference* (CBR). In CBV (which is the default mechanism), when a parameter is passed, the value of the actual parameter is computed and copied to the formal parameter; that is, the parameter access path involves a value copy operation. This increases the storage overhead as the actual and formal parameters occupy different locations. During the execution of the subroutine, all references to the formal parameter uses only the location allocated for it; that is, the content of the location that holds the actual parameter is not modified. In fact, the said location remains unmodified even after the subroutine returns. In a sense, there is no parameter return path. The CBV mechanism is an excellent parameter-passing method for the cases that require only one-way communication (i.e., from caller to callee).

In comparison, in the CBR mechanism, the actual parameter and the formal parameter share the same location.[1] At subroutine invocation time, a pointer to the actual parameter is passed to the subroutine. This parameter access path speeds up the subroutine invocation. However, during subroutine execution, each access to the formal parameter is slower as it requires a level of indirection. Also, CBR can lead to subtle aliasing problems [9]. Since a pointer, not actual value, is passed to the program, the parameter access path and parameter return path are the same, and when the subroutine returns, the actual parameter mirrors the latest update to the formal parameter. In C, programmers enforce the CBR mechanism by passing the address of the actual parameter. It should also be mentioned that arrays are always passed using CBR as using CBV would incur a tremendous copy overhead during subroutine invocation time.

Comparing the overheads incurred by these mechanisms during parameter passing, we observe that CBR is more efficient than CBV in terms of both time (it does not involve memory copy) and space (it does not duplicate data). It should be emphasized, however, that both these mechanisms passes all parameters (i.e., values in CBV and pointers in CBR) at subroutine invocation time. While this uniform treatment of parameters makes the implementation simpler, it may also cause some inefficiencies in cases

[1]Note that in reality C language does not implement CBR directly; it mimics CBR by allowing to pass pointers as values in CBV.

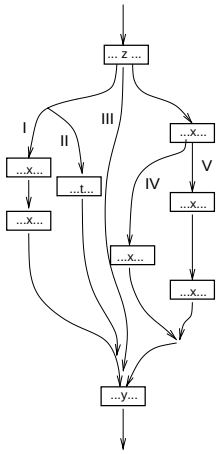

Figure 1. An example control flow graph (CFG).

where the parameter passed is not used in the subroutine. This is particularly problematic if the computation of the actual parameter involves an expensive expression evaluation, an array element computation, or a subroutine call itself (which occur in many large applications). As an example, consider the scenario that occurs in `wood` (one of our applications), where an actual parameter is `u[2i+1][j+k]*2v`, where `u` is a two-dimensional array and `v` is a scalar variable. In order to pass this parameter, the CBV mechanism first computes its value. It should be noted that this process includes (i) computing the values of subscript expressions, (ii) computing array index value using these subscript expressions, (iii) computing the value of `2v`, and (iv) executing the multiplication operation. After computing the value of this actual parameter, the CBV allocates a memory location (for the formal parameter) and then copies this value to that location. Surprisingly, in this case, the CBR mechanism is as costly as CBV. More specifically, what CBR needs to do here is to compute the value of the actual parameter using the same four steps mentioned above and then store the result in some location and pass the address of this location (as a pointer) to the called subroutine. In fact, the overhead of the computation of the actual parameter above can even be much higher if references to `u` or `v` miss in the cache. If this parameter is not used in the called subroutine, both CBV and CBR waste energy as well as execution cycles. This magnitude of this penalty is multiplied if the parameter-passing occurs within a nested loop. Our objective in this paper is to eliminate the energy and performance overhead incurred in such cases using some help from compiler.

3. On-Demand Parameter Passing

3.1. Approach

As noted in the previous section, early evaluation of an unused formal parameter can lead to performance and en-

ergy loss. In this section, we describe an energy-efficient *on-demand* parameter-passing strategy. In this strategy, the value of an actual parameter is not computed unless it is necessary. Note that this not only eliminates the computation for the value of the parameter, but also all computations that lead to that value (and to nothing else). In developing such a strategy, we have two major objectives. First, if the parameter is not used in the called subroutine, we want to save energy as well as execution cycles. Second, if the parameter is actually used in the subroutine, we want to minimize any potential negative impact (of on-demand parameter-passing) on execution cycles.

We discuss our parameter-passing strategy using the control flow graph (CFG) shown in Figure 1. Suppose that this CFG belongs to a subroutine. It is assumed that `x`, `y`, `z`, and `t` are formal parameters and the corresponding actual parameters are costly to compute from the energy perspective. Therefore, if it is not necessary, we do not want to compute the actual parameter and perform pointer (in CBR) or data value (in CBV) passing. We also assume that `...x...` in Figure 1 denotes the use of the formal parameter `x` (and similarly for other variables). It is assumed that these variables are not referenced in any other place in the subroutine. In this CFG, CBV or CBR strategies would compute the corresponding actual parameters and perform pointer/value passing before the execution of the subroutine starts. Our energy-conscious strategy, on the other hand, postpones computing the actual parameters until they are actually needed.

We start by observing that the CFG in Figure 1 has five different potential execution paths from start to end (denoted using `I`, `II`, `III`, `IV`, and `V` in the figure). However, it can be seen that not all formal parameters are used in all paths. Consequently, if we compute the value of an actual parameter before we start executing this subroutine and then execution takes a path which does not use the corresponding formal parameter, we would be wasting both energy and execution cycles. Instead, we can compute the actual parameter on-demand (i.e., only if it really needed). As an example, let us focus on the formal parameter `t`. As shown in the figure, this parameter is used only in path `II`. So, it might be wiser to compute the corresponding actual parameter only along this path (e.g., just before the parameter is used). When we consider formal parameter `z`, however, we see that this parameter is used as soon as the subroutine is entered. Therefore, it needs to be computed when the subroutine is entered, which means that it does not benefit from on-demand parameter passing. A similar scenario occurs when we focus on formal parameter `y`. This parameter is used at the very end of the subroutine where all paths merge. Consequently, the corresponding actual parameter needs to be computed irrespective of the execution path taken by the subroutine. Here, we have two options. We can either compute that actual parameter as soon as the subroutine is entered; or, we can postpone it and compute just before it needs to be accessed (at the very end of the subroutine). Our current implementation uses the latter alternative for uniformity. Accesses to parameter `x` present a more interesting scenario. This variable is accessed in all but two paths. So, if the execution takes path `II` or `III`, the on-demand parameter-passing strategy can save energy. If the execution takes any other path, however, we need to compute the actual parameter. A straightforward implementation would compute the values of actual parameter in six different places (one per each use). As will be discussed later in this paper, a careful implementation can reduce the number of these computations to three, one per path. It should be noted, however, that even these three evaluations (i.e., one per path) present a code size overhead over CBV or CBR parameter-passing strategies (as both CBV and CBR perform a single evaluation per parameter). Therefore, to be fair in comparison, this increase in code size should also be accounted for.

3.2. Global Variables

In this subsection, we show that global variables present a difficulty for on-demand parameter-passing. Consider the following subroutine fragment, assuming that it is called using `foo(c[index])`, where `c` is an array and `index` is a global variable:

```
foo(int x)
 {
  int y;
  ...
  if (...){
   ...
   index++;
   ...
   y=x+1;
   ...}
  else {
   ...
  }
 }
```

It can be seen from this code fragment that a normal parameter-passing mechanism (CBR or CBV) and our on-demand parameter-passing strategy might generate different results depending on which value of the global variable `index` is used. In on-demand parameter evaluation, the actual parameter is computed just before the variable `x` is accessed in statement `y=x+1`. Since computing the value of `c[index]` involves `index` which is modified within the subroutine (using statement `index++`) before the parameter computation is done, the value of `index` used in on-demand parameter-passing will be different from that used in CBR or CBV. This problem is called the *global variable problem* in this paper and can be addressed at least in three different ways:

• The value of `index` can be saved before the subroutine starts its execution. Then, in evaluating the actual parameter (just before `y=x+1`), this saved value (instead of the current value of `index`) is used. In fact, this is the strategy adopted by some functional languages that use lazy evaluation [9]. These languages record the entire execution environment of the actual parameter in a data structure (called *closure*) and pass this data structure to the subroutine. When the subroutine needs to access the formal parameter, the corresponding actual parameter is computed using this closure. While this

strategy might be acceptable from the performance perspective, it is not very useful from the energy viewpoint. This is because copying the execution environment in a data structure itself is a very energy-costly process (in some cases, it might even be costlier than computing the value of the actual parameter itself).

• During compilation, the compiler can analyze the code and detect whether the scenario illustrated above really occurs. If it does, then the compiler computes the actual parameter when the subroutine is entered; that is, it does not use on-demand parameter-passing. In cases the compiler is not able to detect for sure whether this scenario occurs, it conservatively assumes that it does, and gives up on on-demand parameter-passing.

• This is similar to the previous solution. The difference is that when we detect that the scenario mentioned above occurs, instead of dropping the on-demand parameter-passing from consideration, we find the first statement along the path that assigns a new value to the global variable and execute the actual parameter evaluation just before that statement. For example, in the code fragment given above, this method performs the actual parameter computation just before the `index++` statement.

It should be mentioned that Algol introduced a parameter-passing strategy called *call-by-name* (CBN) [9]. When parameters are passed by call-by-name, the actual parameter is, in effect, textually substituted for the corresponding formal parameter in all its occurrences in the subprogram. In a sense, CBN also implements lazy binding. However, there is an important difference between our on-demand parameter-passing strategy and CBN. In CBN, the semantics of parameter passing is different from that of CBV and CBR. For example, in the subprogram fragment given above, the CBN mechanism uses the new value of `index` (i.e., the result of the `index++` statement) in computing the value of the actual parameter (and it is legal to do so). In fact, the whole idea behind CBN is to create such flexibilities where, in computing the values of actual parameters, the effects of the statements in the called subroutine can be taken into account. In contrast, our on-demand parameter-passing strategy does not change the semantics of CBV/CBR; it just tries to save energy and execution cycles when it is not necessary to compute the value of an actual parameter and the computations leading to it.

3.3. Multiple Use of Formal Parameters

If a formal parameter is used multiple times along some path, this creates some problems as well as some opportunities for optimization. To illustrate this issue, we consider the uses of the parameter `x` in path `I` of Figure 1. It is easy to see that this parameter is used twice along this path. Obviously, computing the value of the corresponding actual parameter twice would waste energy as well as execution cycles. This problem is called the *multiple uses problem* in this paper. To address this problem, our strategy is to compute the value of the actual parameter in the first use, save this value in some location, and in the second access to `x`, use this saved value. Obviously, this strategy tries to reuse the previously-computed values of actual parameters as much as possible.

Depending on the original parameter-passing mechanism used (CBV or CBR), we might need to perform slightly different actions for addressing the multiple uses problem. If the original mechanism is CBV, the first use computes the value and stores it in a new location, and the remaining uses (on the same path) use that value. If, on the other hand, the original mechanism is CBR, the first use computes the value (if the actual parameter is an expression), stores it in a location, and creates a pointer which is subsequently used by the remaining uses (on the same path) to access the parameter. In either case, the original semantics of parameter-passing is maintained.

3.4. Problem Formulation and Solution

To perform analyses on a program, it is often necessary to build a control flow graph (CFG). Each statement in the program is a node in the control flow graph; if a statement can be followed by another statement in the program, there is an edge from the former to the latter in the CFG [1]. In this paper, the CFG nodes are individual statements, whereas in most data-flow analysis problem, a CFG nodes contain a sequence of statements without branch. Note that CFG can contain one or more loops as well. In the following discussion, we use the terms *node*, *block*, and *statement* interchangeably. Each node in the CFG has a set of out-edges that lead to *successor nodes*, and in-edges that lead to *predecessor nodes*. The set $pred(b)$ represents all predecessor nodes for statement b and the set $succ(b)$ denotes all successors of b [1].

Data-flow analysis is used to collect data-flow information about program access patterns [1]. A data-flow analysis framework typically sets up and solves systems of equations that relate information at various points in a program (i.e., in various points in the corresponding CFG). Each point of interest in the code contributes a couple of equations to the overall system of equations. In our context, data-flow equations are used to decide the points at which the actual parameter evaluations for a given formal parameter need to be performed. We define a function called $USE()$ such that $USE(b, \mathtt{x})$ returns true if statement (basic block) b uses variable `x`; otherwise, it returns false. Using the $USE()$ function, we make the following definition:

$$EVAL(b, \mathtt{x}) = \begin{cases} \text{true}, & if\ USE(b, \mathtt{x}) \text{ and} \\ & \exists p \in pred(b)\ !EVAL(p, \mathtt{x}) \\ \text{true}, & if\ !USE(b, \mathtt{x}) \text{ and} \\ & \forall p \in pred(b)\ EVAL(p, \mathtt{x}) \\ \text{false}, & \text{otherwise} \end{cases} \quad (1)$$

In this formulation, p denotes a predecessor statement for b. For a given statement b and formal parameter `x` where `x` is used in b, $EVAL(b, \mathtt{x})$ returns true if and only if an actual parameter computation corresponding to the formal parameter `x` needs to be performed to access `x` in b. Such a parameter evaluation would be required if and only if there exists at least a path (coming to statement b) along which the actual parameter evaluation in question has not been

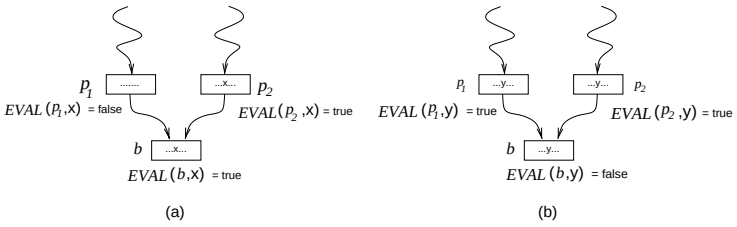

Figure 2. Two different scenarios for $EVAL()$ computation.

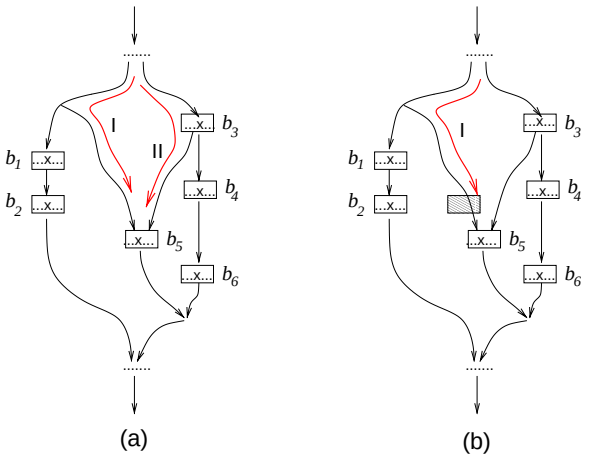

Figure 3. (a) An example CFG. (b) Inserting a basic block for parameter evaluation.

performed yet. As an example, suppose that statement b has two predecessors: p_1 and p_2. Assume that a formal parameter x is used in b and p_2, but not used in p_1; that is, $USE(b, \mathtt{x})$, $USE(p_1, \mathtt{x})$, $USE(p_2, \mathtt{x})$ return true, false, and true, respectively. Assuming that no statement along the path starting from the beginning of the subroutine to p_1 and no statement along the path starting from the beginning of the subroutine to p_2 use the parameter x. In this case, $EVAL(p_1, \mathtt{x})$ computes false and $EVAL(p_2, \mathtt{x})$ computes true. This indicates that the actual parameter evaluation has been performed along the path that contains p_2 but not along the path that contains p_1. Since statement b can be accessed through p_1 or p_2, we need to (conservatively) perform the evaluation of actual parameter in b (see Figure 2(a)); that is, $EVAL(b, \mathtt{x})$ will return true. Suppose now that another formal parameter, y, is used by all these three statements: b, p_1, and p_2. In this case, both $EVAL(p_1, \mathtt{y})$ and $EVAL(p_2, \mathtt{y})$ return true. Assuming that p_1 and p_2 are the only statements through which b can be reached, $EVAL(b, \mathtt{y})$ will return false. Note that in this last scenario when execution reaches b, it is guaranteed that the value of the actual parameter (corresponding to y) has been computed (see Figure 2(b)).

It should be noted that although we also compute the $EVAL()$ function even for the statements that do *not* access the formal parameter in question, the meaning of this function in such cases is different. Specifically, the $EVAL()$ function for such statements is used only for conveying the value (of $EVAL()$) from the previous statements (if any) that use the formal parameter to the successor statements that use it. In technical terms, suppose that b' is a statement that does not access the formal parameter x, and p_1, p_2, ..., p_k are its predecessor statements. If there is at least an i such that $1 \leq i \leq k$ and $EVAL(p_i, \mathtt{x})$ is false, then $EVAL(b', \mathtt{x})$ is set to false; otherwise, it is true.

It should be emphasized that, using $EVAL()$, we can also place actual parameter evaluation code into the subroutine. More specifically, for a given statement b and formal parameter x, we have four possibilities:

- $USE(b, \mathtt{x})$ = true and $EVAL(b, \mathtt{x})$ true: In this case, the actual parameter needs to be computed to access the formal parameter. This means that there exists at least one path from the beginning of the subroutine to b which does not contain the actual parameter computation in question.
- $USE(b, \mathtt{x})$ = true and $EVAL(b, \mathtt{x})$ false: This case means that the statement b needs to access the formal parameter x and the value of the corresponding actual parameter has been computed earlier. Consequently, it does not need to be recomputed; instead, it can be used from the location where it has been stored.
- $USE(b, \mathtt{x})$ = false and $EVAL(b, \mathtt{x})$ false: In this case, the statement b does not use x and is not involved in the computation of the value of the corresponding actual parameter.
- $USE(b, \mathtt{x})$ = false and $EVAL(b, \mathtt{x})$ true: This case is the same as the previous one as far as inserting the actual parameter evaluation code is concerned. No action is performed.

It should be noted that the $EVAL()$ function can be computed in a single traversal over the CFG of the subroutine. The evaluation starts with the header statement h, assuming $EVAL(h, \mathtt{x}) = \emptyset$ for each formal parameter x. It then visits the statements in the CFG one-by-one. A statement is visited if and only if all of its predecessors have already been visited and their $EVAL()$ functions have been computed. These values are used in computing the value of the $EVAL()$ function of the current statement using the expression (1) given above. While it is possible to compute the $EVAL()$ function of all variables simultaneously in a single traversal of the CFG, our current implementation performs a separate traversal for each variable. This is a viable option as the number of formal parameters for a given subroutine is generally a small number.

We now give an example to explain how our approach handles a given formal parameter. Consider the CFG in Figure 3(a), assuming that b_1, b_2, b_3, b_4, b_5, and b_6 are the only statements that use formal parameter x. We start by observing that $EVAL(b_1, \mathtt{x})$ should be true as there is no way that the value of the actual parameter might be required before reaching b_1 (along the path that leads to b_1). Having computed $EVAL(b_1, \mathtt{x})$ as true, it is easy to see that $EVAL(b_2, \mathtt{x})$ should be false as b_1 is the only predecessor to b_2. Informally, what this means is that, since we compute the value of the actual parameter in b_1, we do not need to recompute it in b_2. In a similar fashion, it can easily be seen

```
INPUT: A CFG with USE(b, x) computed
   for each basic block b and formal parameter x
OUTPUT: The computed EVAL(b, x) function

 for each node b in CFG do
  b.processed ←— false;
 endfor;
 B ←— nodes in CFG without predecessor;
 while (B ≠ ∅)
  remove a node b from B
  compute EVAL(b, x) using the expression (1);
  b.processed ←— true;
  for each s ∈ succ(b) do
   if ∀p ∈ pred(s) p.processed = true then
    if s.processed = true then
     B ←— B ∪ s;
  endfor;
 endwhile;
```

Figure 4. Algorithm to compute $EVAL()$.

that $EVAL(b_3, \mathtt{x})$, $EVAL(b_4, \mathtt{x})$, $EVAL(b_6, \mathtt{x})$ should be true, false, and false. The statement b_5 presents an interesting case. Since this statement can be reached through two different paths (shown as I and II in the figure), in deciding what $EVAL(b_5, \mathtt{x})$ should be, we need to consider both the paths. If b_5 is reached through b_3, we can see that no actual parameter computation (in b_5) is necessary. If, however, it is reached through path I, we need to compute the value of the actual parameter. Consequently, we conservatively determine that $EVAL(b_5, \mathtt{x})$ is true; that is, the value of the actual parameter should be computed before the formal parameter is accessed in b_5.

Figure 4 gives the data-flow algorithm for computing the $EVAL()$ function for each statement in a given subprogram. The set B in this algorithm holds the basic blocks (statements) that can be processed; that is, the ones whose all predecessors have already been computed. Assuming that we have S subroutines, the overall optimization approach is bounded by $O(SVM^2)$, where V is the maximum number of formal parameters to any subroutine in the application and M is the maximum number of statements in any subroutine.

3.5. Additional Optimization

In data-flow analysis, we generally accept safe (or conservative) solutions which might, in some cases, lead to inefficiencies. In our case, the $EVAL()$ computation algorithm presented in the previous subsection might recompute the value of an actual parameter more than once when different execution paths merge at some CFG point. For example, consider the statement b_5 in Figure 3(a). If, during execution, path II is taken, then the value of the actual parameter corresponding to x will be computed twice, wasting energy as well as execution cycles. In this work, we consider two different methods to handle this problem. In the first method, called *block insertion*, we create a new basic block and put the actual parameter computation code there. Figure 3(b) illustrates this solution, which creates a new block (shown shaded) and inserts it along path I just before the statement b_5 is reached. The actual parameter evaluation is performed in this new basic block and the statement b_5 does not perform that evaluation. In this way, we guarantee that when b_5 is reached, the actual parameter has already been computed; so, b_5 can just use the value.

The second method, called *path control*, is based on the idea of using a variable (or a set of variables) to determine at runtime which path is being executed. For example, assume that we use a variable p to determine which path (leading to b_5) is being executed. Without loss of generality, we can assume that if path I is taken p is assigned 1, otherwise p is set to zero. Under this method, when b_5 is reached, we can check the value of p, and depending on its value, we perform actual parameter value evaluation or not. It should be noted that, as compared to the first method, this method is expected to result in a smaller executable size (in general); but, it might lead to a higher execution time due to comparsion operation.

4. Conclusions

Embedded systems demand energy efficiency in order to maximize the battery life. While previous work has concentrated on reducing energy consumption using circuit, architecture, and OS level techniques, in this work, we studied the possibility of modifying the parameter-passing mechanism of the language with some help from compiler. Using a set of five real-life applications and a custom simulator, we investigated the energy and performance impact of an on-demand parameter-passing strategy. In this strategy, the value of an actual parameter is not computed if the corresponding formal parameter is not used within the subroutine.

References

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*, Addison-Wesley Publishing Company, 1986.

[2] A. Chandrakasan, W. J. Bowhill, and F. Fox. *Design of High-Performance Microprocessor Circuits*. IEEE Press, 2001.

[3] L. Benini and G. De Micheli. System-level power optimization: techniques and tools. *ACM Transactions on Design Automation of Electronic Systems*, 5(2), pp.115-192, April 2000.

[4] E.-Y. Chung, L. Benini, and D. De Micheli. Energy-efficient source code transformation based on value profiling. In *Proc. the 1st Workshop on Compilers and Operating Systems for Low Power*, Philadelphia, PA, 2000.

[5] E.-Y. Chung, L. Benini, and D. De Micheli. Automatic source code specialization for energy reduction. In *Proc. the International Symposium on Low Power Electronics and Design*, Huntington Beach, CA, August 2001.

[6] K. Cooper, M. Hall, and K. Kennedy. A methodology for procedure cloning. *Computer Languages*, Vol 19, No 12, pp. 105-117, April 1993.

[7] M. Kamble and K. Ghose. Analytical energy dissipation models for low power caches. In Proc. *the International Symposium on Low Power Electronics and Design*, page 143, August 1997.

[8] G. Lakshminarayana, A. Raghunathan, K. Khouri, K. Jha, and S. Dey. Common-case computation: a high-level technique for power and performance optimization. In Proc. *the Design Automation Conference*, pp. 56-61, 1999.

[9] R. W. Sebesta. *Concepts of Programming Languages*, Addison-Wesley Publishing, 2001.

[10] V. Tiwari, S. Malik, and A. Wolfe. Power analysis of embedded software: a first step towards software power minimization. *IEEE Transactions on Very Large Scale Integration Systems*, 2(4), December 1994.

[11] M. Weiser, B. Welch, A. Demers, and S. Shenker. Scheduling for reduced CPU energy. In *Proc. the USENIX Symposium on Operating Systems Design and Implementation*, 1994, pp. 13–23.

[12] W. Wang, A. Raghunathan, G. Lakshminarayana, and N. K. Jha. Input space adaptive design: a high-level methodology for energy and performance optimization. In *Proc. the Design Automation Conference*, Las Vegas, NV, 2001.