BB-GC: Basic-Block Level Garbage Collection

Özcan Özturk, Mahmut Kandemir, Mary Jane Irwin

BB-GC: Basic-Block Level Garbage Collection*

Ozcan Ozturk, Mahmut Kandemir, and Mary Jane Irwin
Computer Science and Engineering Department
The Pennsylvania State University, University Park, PA 16802, USA
{ozturk, kandemir and mji}@cse.psu.edu

Abstract

Memory space limitation is a serious problem for many embedded systems from diverse application domains. While circuit/packaging techniques are definitely important to squeeze large quantities of data/ instruction into small size memories typically employed by embedded systems, software can also play a crucial role in reducing memory space demands of embedded applications. This paper focuses on a software-managed two-level memory hierarchy and instruction accesses. Our goal is to reduce on-chip memory requirements of a given application as much as possible, so that the memory space saved can be used by other simultaneously-executing applications. The proposed approach achieves this by tracking the lifetime of instructions. Specifically, when an instruction is dead (i.e., it could not be visited again in the rest of execution), we deallocate the on-chip memory space allocated to it. Working on the control flow graph representation of an embedded application, our approach performs basic block-level garbage collection for on-chip memories.

1. Introduction and Motivation

Embedded or real-time systems include all those in which constrains imposed by the environment (e.g., available battery power and memory capacity) play a critical role in the design and implementation of the system. Common areas for embedded systems are machine and process control, medical instruments, smart telephony and data acquisition, and memory limitation is a serious concern in all these areas. In addition, as embedded systems become increasingly complex, there is a growing demand for executing multiple applications concurrently, thereby putting even higher pressure on memory system.

A common solution to the memory problem is to design a memory hierarchy where the higher levels are small, fast, and energy efficient, and the lower levels are large, slow, and energy hungry. A frequent instantiation of this model is a two-level hierarchy with an on-chip cache memory (possibly separate components for instructions and data) and an off-chip main memory. However, there are several critical problems associated with conventional hardware-managed caches. First, the success of a cache-based system depends strongly on the compatibility between the data access pattern and the hardware-guided cache line replacement policy. In many cases, the cache line-level data management can be very fine granular for a data-intensive embedded application. Second, since data transfers in and out of the cache are managed by hardware, there is no guarantee that a required data item will be in the cache at the time of access. This may be an important problem in real-time embedded systems where execution time predictability is a critical issue. Third, in many situations, the cache management policy is too general for a given embedded application. Fourth, the dynamic mapping of data to cache locations can be costly from an energy consumption viewpoint. These problems have motivated recent research on software-managed on-chip memories [9, 10], where the data/instruction transfers between the on-chip and off-chip memories are controlled by the software (e.g., compiler).

In such a software-controlled two-level memory hierarchy, one of the critical issues is ensuring effective use of available on-chip memory. Specifically, it is both performance and energy efficient to satisfy most of instruction/data requests from the on-chip memory. This is particularly important when available on-chip memory space is shared by multiple applications. There are at least two ways of increasing the effectiveness of on-chip storage:

• *Increasing data reuse.* This helps reduce the frequency and volume of the transfers between the on-chip and off-chip memory. The prior work considered code restructuring and data layout optimizations for improving data reuse in caches and software-managed memories.

• *Reducing memory space requirements.* The studies in this group exploit the lifetime information of different data to check whether two different data can share the same memory locations. As in the first group, the existing techniques target at both caches and software-managed memories.

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

In this paper, we focus on a software-managed two-level memory hierarchy and *instruction accesses*. Our goal is to reduce on-chip (instruction) memory requirements of a given application as much as possible, so that the memory space saved can be used by other applications. We achieve this by tracking the lifetime of instructions at the basic block granularity. Specifically, when an instruction is *dead* (i.e., it could not be visited again in the rest of the execution), we deallocate the on-chip memory space allocated to it. Working on the control flow graph (CFG) representation of the procedure, in a sense, our approach performs *basic block-level garbage collection* for on-chip memories.

Our scheme can be used in two different ways. In a multi-programmed embedded environment, the memory space saved can be made available to other applications, thereby effectively increasing the degree of multi-programming. Alternately, in a multi-bank on-chip memory, the unused banks can be switched off to a low-power mode to save energy. In this paper, we also discuss how careful block placement in on-chip memory can reduce memory fragmentation.

The rest of this paper is structured as follows. Section 2 describes our two-level software-managed memory hierarchy and code execution model under this memory system. Section 3 gives the details of our approach to deallocating the on-chip memory space of dead basic blocks. Section 4 studies our aggressive on-chip memory deallocation where even the space occupied by live basic blocks are recycled for decreasing average memory occupancy further. Section 5 discusses the placement of basic block in on-chip memory to reduce memory fragmentation when dead blocks are deleted. Section 6 discusses related work, and finally, Section 7 summarizes our contributions.

2. Architecture

We assume a two-level software-managed instruction memory hierarchy, where the first level is on-chip and the second level is off-chip. Although our software-managed on-chip instruction memory is similar in performance/energy characteristics to a conventional hardware-managed on-chip cache memory, its management is very different from that of an instruction cache. Unlike instruction caches, the instruction flow to the on-chip instruction memory is controlled by software (compiler in our case). Since the on-chip memory is assumed to be very small in size compared to data volume that needs to be processed, its effective management is critical from both the power and performance perspectives. In particular, on-chip memory space can be shared by multiple applications at the same time, and its effective use determines the number of applications that could be run simultaneously. In this architecture, the on-chip memory space that could be saved/reclaimed from an application can be made available to other applications. We also assume existence of a separate on-chip data memory, but this paper is focused on instruction accesses (the off-chip memory can also be used to store data). Note that the architecture we are focusing on in this study contains a software-managed hierarchy and no hardware-managed cache. This allows the compiler to explicitly manage the data transfers in the memory system and discard the basic blocks whose lifetimes are over.

3. Our Approach

3.1. High-Level View

Our baseline approach operates as follows. When a procedure is invoked, all its basic blocks are transferred from off-chip memory to on-chip memory. As it executes from on-chip memory, we check whether any of its basic blocks is dead at the current point. If this is the case, that basic block is deleted from on-chip memory. In this work, a basic block (of code) is called *dead* if it is not possible that the execution thread can visit it in the rest of the program execution.

As a simple example to illustrate the concept, we consider the CFG depicted in Figure 1(a). When the execution thread reaches the basic block marked 2 we know for sure that the basic blocks marked as 1, 6, 7, and 8 are dead (i.e., they cannot be accessed); therefore, the on-chip memory space occupied by them can safely be deallocated. Similarly, if, on another input, the execution reaches basic block 8, the on-chip space reserved for blocks 6 and 7 can be reclaimed (in addition to the space allocated for blocks 1, 2, 3, 4, and 5, which would be deallocated when basic block 6 is reached). However, when a procedure whose on-chip memory space has been deallocated is later re-invoked, all its basic blocks must be re-brought into the on-chip memory for execution. In the remainder of this paper, this approach of deallocating on-chip storage space of dead basic blocks is referred to as *Basic Block-Level Garbage Collection*, or BB-GC for short. The two subsections below describe BB-GC in detail.

3.2. Basic Block Marking for Space Reclaiming

BB-GC gives the task of managing the contents of the on-chip instruction memory to the compiler. Specifically, the compiler analyzes the application code, which is represented as a set of connected control flow graphs (CFGs) – one per procedure. A section of program code that does not cross any conditional branches, loop boundaries, or other transfers of control is referred to as *basic block* (BB). In a basic block, the branch instruction, if any, can occur only as the last instruction in the block. In a control flow graph, each node represents a basic block, and an edge from a node to another indicates a potential flow of control that could occur at runtime [8]. Note that CFG is a conservative representation since it includes all possible flows of control in the program, some of which could not be exercised for a particular input. Our compiler analysis performs two important tasks:

• It inserts a special assembly instruction before each procedure call to copy the contents of the procedure from the off-chip memory to the on-chip memory, if it is not already there. Before doing so, however, it also deletes any basic blocks (of that procedure) that might have been left in the on-chip memory from the previous invocation. While at first glance this might seem overkill as one may potentially reuse some of these basic blocks, deleting such basic blocks and copying the entire procedure from the off-chip memory make the implementation much simpler than an alternate scheme that could try to track current locations and status of basic blocks at runtime.

• To each basic block BB_i, it attaches a set of basic blocks ids ($\mathcal{B}_i$) observing the following rule: whenever BB_i is reached, all the basic blocks $BB_j \in \mathcal{B}_i$ can be safely declared dead, and the on-chip instruction memory space allocated to them can be returned to the pool of free spaces (this process is also called memory space reclamation). Note that the set $\mathcal{B}_i$ for a given basic block BB_i can be empty. For the example shown in Figure 1(a), we have the following $\mathcal{B}_i$ sets (also called the *dead block sets*):

$$\begin{aligned}
\mathcal{B}_1 &= \emptyset \\
\mathcal{B}_2 &= \{BB_1, BB_6, BB_7, BB_8\} \\
\mathcal{B}_3 &= \emptyset \\
\mathcal{B}_4 &= \emptyset \\
\mathcal{B}_5 &= \emptyset \\
\mathcal{B}_6 &= \{BB_1, BB_2, BB_3, BB_4, BB_5\} \\
\mathcal{B}_7 &= \emptyset \\
\mathcal{B}_8 &= \{BB_6, BB_7\} \\
\mathcal{B}_9 &= \{BB_2, BB_3, BB_4, BB_5\} \text{ or } \{BB_8\}
\end{aligned}$$

We will describe shortly how these sets are automatically determined by BB-GC. The compiler also inserts an instruction at the beginning of block BB_i to delete all the basic blocks in $\mathcal{B}_i$ from the on-chip memory (if they have not been already deleted). Four important points should be noted here. First, the instruction to delete the blocks in $\mathcal{B}_i$ should attempt such a deletion only in its first invocation. For example, considering the CFG in Figure 1(a) again, when the execution arrives at the basic block marked 6 the first time, we need to delete blocks BB_1, BB_2, BB_3, BB_4, and BB_5. In the potential subsequent visits of this block (via the edge between BB_7 and BB_6), we should not attempt to delete any basic blocks (as the blocks in question would have already been deleted, and their space could have been re-assigned to other applications). This checking incurs an extra overhead which is accounted for in our experimental evaluation. The second point is that the determination of the $\mathcal{B}_i$ sets should be done with care to minimize the overheads. For example, in Figure 1(a), $\mathcal{B}_3$ is $\emptyset$ since when BB_2 is visited first time, all four blocks (BB_1, BB_6, BB_7, and BB_8) would be deleted, and no additional basic blocks can be deleted when BB_3 is visited. The third point is that, due to lack of complete control flow information at compile time, the determination of some $\mathcal{B}_i$ sets may need runtime information as well. For example, the basic blocks that will be removed from memory when execution arrives at BB_9 depends on the edge through which the execution enters BB_9. We can keep track of this at runtime by using some extra variables as explained below (again with some extra overhead). Finally, when the last basic block completes its execution, all remaining basic blocks of the procedure are deleted from the on-chip memory. In the rest of this subsection, we discuss the algorithm that generates the $\mathcal{B}_i$ sets.

The proposed algorithm works on a CFG representation of a procedure, and is executed for each procedure in turn. It has two steps. The first step is an iterative loop, and in each iteration of this loop, we first determine the set of nodes (basic blocks) that are *reachable* from a particular node. Let us use $\mathcal{C}_i$ denote the set of such nodes for basic block BB_i. We refer to the set $\mathcal{P}_i$ = $\mathcal{N}$ - $\mathcal{C}_i$ as the *potential dead block set* for BB_i, assuming that set $\mathcal{N}$ contains all the basic blocks in the procedure under consideration. We call this set as "potential" since the second step of our algorithm eliminates some of these sets and/or reduces the number of elements (basic blocks) in some others. To determine the set $\mathcal{C}_i$, one can use either depth-first search (where we search as deeply as possible by visiting a node, and then recursively performing depth-first search on each adjacent node) or breadth-first search (where we search as broadly as possible by visiting a node, and then immediately visiting all nodes adjacent to that node), the two techniques commonly employed in applications that perform graph processing.

In the second step, we use the *dominator* concept from the compiler theory [8]. We say node BB_i of the CFG *dominates* node BB_j if every path from the initial node of the CFG to BB_j goes through BB_i. That is, it is not possible to reach BB_j without first reaching BB_i. Let $\mathcal{D}_i$ represent the set of nodes that are dominated by BB_i, and $\mathcal{D}_i^{-1}$ the set of nodes that dominate BB_i. In the second step of our algorithm, we reduce $\mathcal{P}_i$ to $\mathcal{B}_i$. Initially, for all the basic blocks BB_i without any predecessors in the dominator tree, we set $\mathcal{B}_i = \mathcal{P}_i$. Then, we use the following formula recursively to determine the $\mathcal{B}$ sets of the remaining nodes in the CFG if they are not merge nodes:

$$\mathcal{B}_i = \mathcal{P}_i - \bigcup_k \mathcal{B}_k,$$

where $BB_k \in \mathcal{D}_i^{-1}$ and $\cup$ denotes set union operator. In other words, we exclude from $\mathcal{P}_i$ all the nodes that are already captured by its dominators. On the other hand, for the merge nodes (i.e., the nodes into which multiple edges enter), we use:

$$\mathcal{B}_i = \mathcal{P}_i - \{\bigcup_k \mathcal{B}_k \cup \mathcal{V}_i\},$$

where $\mathcal{V}_i$ is the set of basic blocks that become dead when the execution reaches BB_i but could not be captured by any dominator. It is important to note that a basic block can belong to $\cup_k \mathcal{B}_k$ or $\mathcal{V}_i$ but not both. Note also that, to compute $\mathcal{V}_i$, at runtime, we need to keep track of the CFG edge from which we come to BB_i. We do this by associating a variable

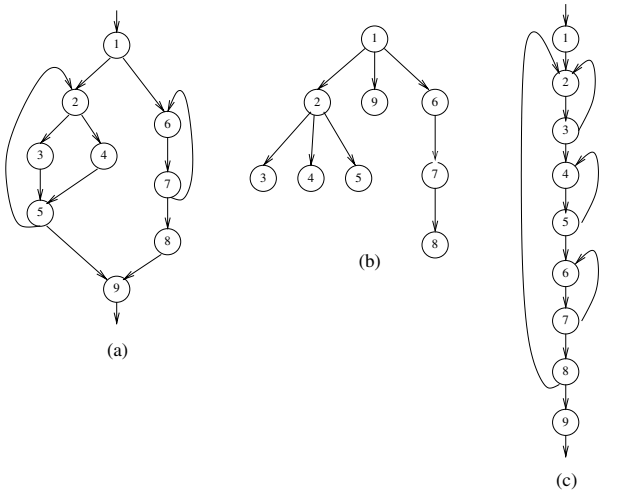

Figure 1. (a-c) Two different CFGs. (b) Dominator tree for the CFG in (a).

with each basic block. If there are f edges entering this basic block, this variable can take f different values (from 0 to $f-1$). The compiler adds special instructions to set the value of this variable for each merge node. Our experimental evaluation also accounts for the space and performance overhead of maintaining these extra variables.

Let us now briefly discuss the complexity of our approach. The asymptotic complexity of the search in the first step is $O(E+N)$, where E is the number of edges in the CFG and N is the number of nodes. Since $N \ll E$ in general, we can simplify this cost as $O(E)$. Therefore, the total cost of the first step is $O(NE)$ since the search is repeated for every node. Finding the dominators in flow graphs can be done in $O(E)$ time in practice [6] (so can determining the $\mathcal{V}_i$ sets), which is the main factor that determines the complexity of the second step of the algorithm. Consequently, the overall complexity of the static part of our approach is $O(NE)$.

Considering the CFG in Figure 1 once more, after the first step, we determine the potential dead block sets as follows:

$$\begin{aligned}
\mathcal{P}_1 &= \emptyset \\
\mathcal{P}_2 &= \{BB_1, BB_6, BB_7, BB_8\} \\
\mathcal{P}_3 &= \{BB_1, BB_6, BB_7, BB_8\} \\
\mathcal{P}_4 &= \{BB_1, BB_6, BB_7, BB_8\} \\
\mathcal{P}_5 &= \{BB_1, BB_6, BB_7, BB_8\} \\
\mathcal{P}_6 &= \{BB_1, BB_2, BB_3, BB_4, BB_5\} \\
\mathcal{P}_7 &= \{BB_1, BB_2, BB_3, BB_4, BB_5\} \\
\mathcal{P}_8 &= \{BB_1, BB_2, BB_3, BB_4, BB_5, BB_6, BB_7\} \\
\mathcal{P}_9 &= \{BB_1, BB_2, BB_3, BB_4, BB_5, BB_6, BB_7, BB_8\}
\end{aligned}$$

Figure 1(b) illustrates the *dominator tree* for the CFG in Figure 1(a). A dominator tree is a convenient way of presenting dominator information, in which the initial node is the root, and each node BB_i dominates only its descendants in the tree. We see from this dominator tree that BB_2 dominates BB_3, BB_4, and BB_5. Consequently, in determining $\mathcal{B}_i$ where $i = 3, 4, 5$, we have $\mathcal{B}_i = \mathcal{P}_i - \{BB_1, BB_6, BB_7, BB_8\}$, which gives us the empty set for all three basic blocks. The dead block set of the other basic blocks in this example can be found using a similar strategy, except for BB_9, which we discuss in more detail now. Note that, for BB_9, $\cup_k \mathcal{B}_k$ is empty set as BB_9 has only single dominator (see Figure 1(b)). If, during execution, we reach BB_9 through BB_5, we need to delete only nodes BB_2, BB_3, BB_4, and BB_5. The reason that we delete only these blocks is that the others must have already been deleted by the time the execution arrives at BB_9 (through BB_5). In other words, $\mathcal{V}_9$ is $\{BB_2, BB_3, BB_4, BB_5\}$. Note that we determine the potential $\mathcal{V}_9$ sets at compile time. Similarly, if the execution comes at BB_9 via BB_8, we have $\mathcal{V}_9 = \{BB_8\}$. Again, the reason for this is that, when we come to BB_9 via BB_8, the latter is the only node (apart from BB_9 that still occupies memory). Our compiler records these alternate $\mathcal{V}_9$ sets, and depending on the value of the runtime variables mentioned above, it uses (selects) the appropriate $\mathcal{V}_9$ set at runtime. Therefore, at the end of this process, we find the $\mathcal{B}$ sets shown earlier in this subsection.

3.3. Discussion

Our approach presented in the previous subsection has the important characteristic that it transfers an entire procedure body to the on-chip memory when the procedure is invoked. It then tries to reduce the on-chip memory space occupied by the procedure by removing its basic blocks considering their lifetimes. One might point out that an alternate implementation is also possible. Specifically, instead of bringing all the basic blocks at once at the time of invocation, we can bring them on a need basis. In other words, we can manage the on-chip memory as a cache for hot basic blocks, i.e., the ones that are executed frequently. In addition, when necessary, we can also employ a block eviction strategy based on LRU or a similar algorithm. In fact, the strategy described in [2] implements a similar scheme (though, instead of LRU, it relies on loop structures in the CFG). Unfortunately, one of the major consequences of such a scheme is that, at a given time, some of the basic blocks of the procedure can reside in the on-chip memory, whereas the others are in the off-chip memory, and the location of a basic block with respect to its neighbors can change during the course of execution. As a result, each time a basic block changes location (and the number of such location changes can be very high in a typical execution), the target addresses of all the branches to that basic block must be updated for correct execution. This situation is depicted in Figure 2. In Figure 2(a), three basic blocks (BB_2, BB_3, and BB_4) are in the on-chip memory, whereas in Figure 2(b) two basic blocks (BB_2 and BB_4). Notice that, depending on where basic block BB_3 is, the target of the branches to it from basic block BB_2 will be different (similar argument goes for block BB_4 as well, which also branches to block BB_3). This type of dy-

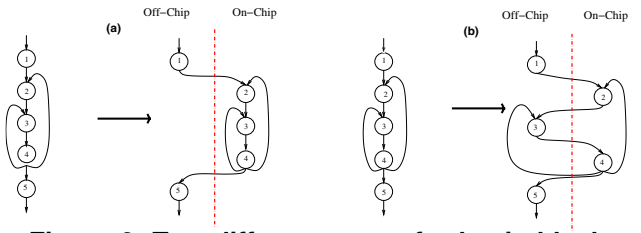

Figure 2. Two different cases for basic block level on-chip memory management.

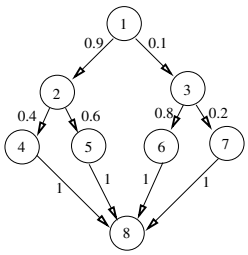

Figure 3. An example CFG with branch probabilities.

namic address management at runtime is very difficult to track, is time consuming, and makes the underlying implementation very complex. This complexity can in turn have negative impacts on execution cycles and power consumption of the entire design. In contrast, in our proposal, all the intra-procedure branches are within the same memory component (on-chip), and consequently, the address management is much simpler. We rely on the fact that most of the branches among the basic blocks are within procedures not across the procedures. The basic block branches across procedures correspond to procedure calls and are relatively rare.

4. Aggressive Memory Deallocation

It is to be noted that the approach discussed in Section 3 is a conservative one. That is, the on-chip memory space allocated to a basic block is deallocated only when the block is dead, i.e., there is no CFG arch using which the block can be re-visited. While this conservative strategy eliminates any potential negative impact of space deallocation on execution cycles, one could do even better from a pure memory space saving perspective if one could deallocate blocks before they become dead. The main potential drawback of this strategy is the increase in execution cycles due to increased traffic between the on-chip and off-chip memories.

While it is possible to adopt several strategies to decide which basic blocks to deallocate, any viable scheme must observe the rule that the space of a basic block that will be re-accessed (executed) shortly should not be deallocated (as this can significantly increase the volume of transfers between the off-chip and on-chip memory). In other words, one needs to select the victim block(s) among the ones with large inter-access time gap. An important question then is how one can decide whether the next access to a basic block is really far in execution. The scheme proposed in this section assumes that the next access to a given basic block can be predicted to be far if between the current access and next access there is at least a *loop* to be executed. The rationale behind this assumption is that most loops iterate large number of times before they terminate. Consequently, during long loop execution, one can achieve large savings in average memory occupancy.

We expect this scheme to be most effective in applications whose CFGs have a big loop surrounding most of the basic blocks. Consider, for example, the CFG depicted in Figure 1(c). An important difference between this CFG and the one in Figure 1(a) is that former has a large outermost loop, whose back edge goes from BB_8 to BB_2. The scheme described in the previous section cannot be very successful in this CFG as most of the basic blocks are dead only after basic block BB_8 exits (which is too late to save any memory space). In contrast, the aggressive scheme can be effective in such scenarios. Consider, for example, the loop that consists of blocks BB_2 and BB_3 in this CFG. The aggressive scheme deallocates the on-chip memory space allocated to them after the loop exits, i.e., the execution takes the edge between BB_3 and BB_4. The reason is that, when BB_4 is reached, re-visiting BB_2 will require two loops (i.e., BB_4-BB_5 and BB_6-BB_7) to be completed. In our experimental evaluation, we compare our baseline implementation with this aggressive dead block elimination scheme.

5. Memory Placement of Basic Blocks

One of the characteristics of BB-GC is that it deallocates a memory region (that holds a dead basic block or a block whose next invocation is far in execution) whenever it is possible to do so. As a consequence, it is possible that the instruction memory space is fragmented as the execution progresses. Unfortunately, it is well known [5] that a fragmented memory (where numerous small free regions/chunks are distributed all over the memory space) makes it difficult to utilize it for allocating space for new data/code structures. In other words, one would prefer a few large free regions in memory instead of many small free regions. In this section, we discuss a strategy, using which basic blocks of a procedure can be mapped to the on-chip instruction memory such that the deallocations generate large free spaces in memory as much as possible.

To achieve this, the proposed approach makes use of *profile data.* The specific profile information we use is the execution probabilities for CFG edges. Using these probabilities, our approach, which is an iterative process, decides how the basic blocks of the procedure should be laid out in memory.

Algorithm 1 $ProfileBasedSearch(v, L)$

```
1: put v to the end of L.
2: while there are unvisited edges incident on v do
3:    find e with smallest probability among unvisited edges incident on v.
4:    find w such that e = (v, w)
5:    if all incoming edges to w are visited then
6:       ProfileBasedSearch(w, L)
7:    end if
8:    mark e as visited
9: end while
```

The approach can be best explained using an example. Consider the CFG illustrated in Figure 3. The numbers attached to the edges represent branch probabilities. As an example, the probability with which the execution moves to BB_2 after exiting block BB_1 is 90%, whereas that of proceeding with BB_3 (again, after exiting BB_1) is 10%. Based on these probabilities, we can decide that it is better to store BB_3 and BB_1 consecutively in memory since it is more likely that they will be dead at the same time, and (when this happens) their space can be deallocated together (which results in a larger free space). After making this decision, our approach continues with the next (unprocessed) smallest probability edge as deep as possible in the CFG. In our example, the next basic block to be stored (next to BB_3 and BB_1) is selected based on the probabilities of the edges outgoing from BB_3. With a probability of 20%, BB_7 would be the next one in the list. After this, we consider BB_8 since it is the only candidate that could be reached from BB_7. However, there exist other (unprocessed) edges entering BB_8. As it can be seen from the CFG that it is not possible to deallocate the space for BB_8 until the entire procedure terminates. Consequently, our algorithm checks whether all of the incoming edges of a basic block are visited before it is decided to be stored next to the blocks that have already been processed. Therefore, after having reached BB_8, our algorithm backtracks to BB_7 and then BB_3. The next smallest probability edge would lead to basic block BB_6 and this shapes our current list of blocks to be stored next to each other as BB_1, BB_3, BB_7, BB_6. Since BB_8 still has unvisited incoming edges, our algorithm backtracks to BB_1. In a similar fashion, we compute the next set of basic blocks to be stored one next to another as BB_2, BB_4 and BB_5. Finally, BB_8 would be the last basic block in the list since the edge between BB_5 and BB_8 is the last unvisited edge incoming to BB_8. So, the final storage order would be BB_1, BB_3, BB_7, BB_6, BB_2, BB_4, BB_5, BB_8. In comparison, if we were to use a depth first search (DFS) algorithm to determine the storage order for basic blocks, it would be as follows: $BB_1, BB_2, BB_4, BB_8, BB_5, BB_3, BB_6, BB_7$. And, similarly, a breadth first search (BFS) algorithm would result in BB_1, BB_2, BB_3, BB_4, BB_5, BB_6, BB_7, BB_8. Algorithm 1 gives the sketch of our algorithm for determining the order in which basic blocks are laid out in memory. A call to $ProfileBasedSearch(s, L)$ forms the layout: $L[0] \ldots L[|v|]$, where $|v|$ is the number of basic blocks and s is the first basic block in the CFG.

6. Related Work

Several prior studies focused on the use of scratch-pad memories (SPMs) for data accesses. For example, Panda et al. [9] present a powerful static data partitioning scheme for efficient utilization of scratch-pad memory. Cooper and Harvey [3] present two compiler-directed methods for software-managing a small cache for holding spilled register values. Hallnor and Reinhardt [4] propose a new software-managed cache architecture and a new data replacement algorithm. Bellas et al. [1] present an SPM management scheme for instruction accesses. Steinke et al. [10] focus on a strategy for placing program and data objects into SPM for saving energy. Lee et al. [7] also focus on reducing the energy consumption due to instruction accesses using a software-managed SPM (called loop cache).

7. Concluding Remarks

One of the biggest roadblocks preventing software writers to achieve higher performance in embedded environments is memory limitation. Consequently, utilizing available on-chip memory space in the most effective way has been one of the most attractive research topics. Focusing on on-chip memory and instruction accesses, this paper presents a scheme that reduces the average memory occupancy of instructions by reclaiming the space allocated to dead basic blocks, i.e., basic blocks that completed their last execution. The paper also studies a more aggressive scheme and discusses the trade-offs between memory savings and performance overhead. In addition, we study the impact of basic block placement in memory.

References

[1] N. Bellas, I. N. Hajj, C. Polychronopoulos, and G. Stamoulis. Energy and performance improvements in microprocessor design using a loop cache. In Proc. ICCD, 1999, pp. 378–383.

[2] G. Chen et al. Compiler-directed management of instruction accesses. In Proc. Euromicro Symposium on Digital System Design, Architectures, Methods and Tools, Turkey, September, 2003.

[3] K. D. Cooper and T. J. Harvey. Compiler-controlled memory. In Proc. ASPLOS, CA, November 1998.

[4] E. G. Hallnor and S. K. Reinhardt. A fully-associative software-managed cache design. In Proc. International Conference on Computer Architecture, pp. 107–116, Vancouver, British Columbia, Canada, 2000.

[5] R. Jones and R. Lins. Garbage Collection: Algorithms for Automatic Dynamic Memory Management. John Wiley and Sons, 1996.

[6] T. Lengauer and E. Tarjan. A fast algorithm for finding dominators in a flow graph. ACM Transactions on Programming Languages and Systems, Vol.1, No.1, pp.121–141, July, 1979.

[7] L. H. Lee, B. Moyer and J. Arends. Instruction fetch energy reduction using loop caches for embedded applications with small tight loops. In Proc. ISLPED, San Diego, CA, August, 1999.

[8] S. S. Muchnick. Advanced compiler design and implementation, Morgan Kaufmann Publishers, San Francisco, CA, 1997.

[9] P. R. Panda, N. D. Dutt, and A. Nicolau. Efficient utilization of scratch-pad-memory in embedded processor applications.

[10] S. Steinke et al. Assigning program and data objects to scratch-pad for energy reduction. In Proc. DATE'02, Paris, France, 2002.