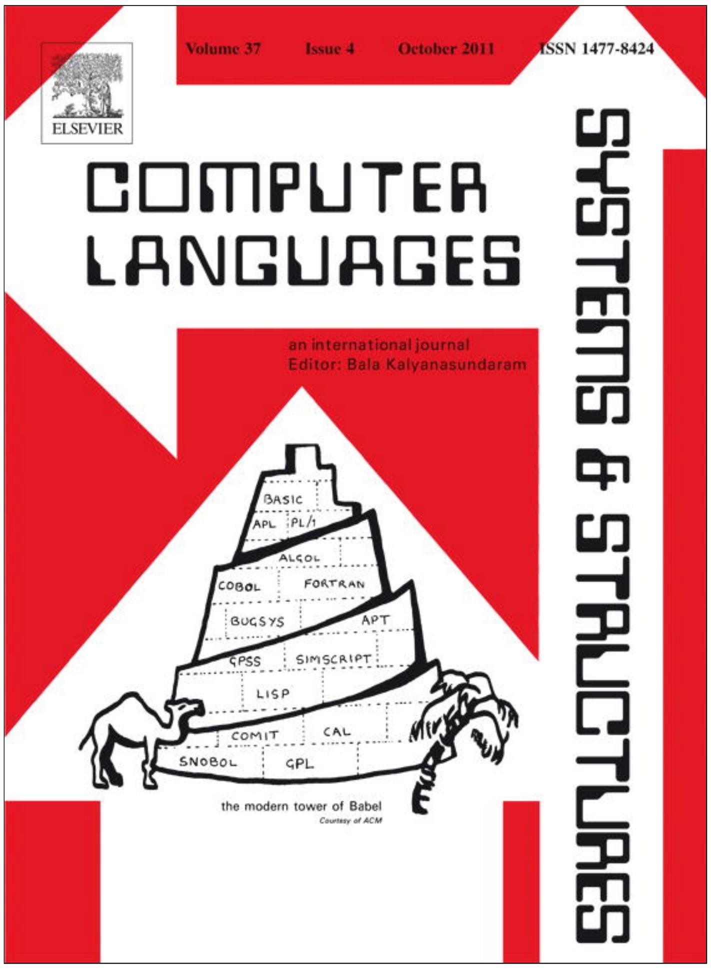

Volume 37 Issue 4 October 2011 ISSN 1477-8424
ELSEVIER
COMPUTER LANGUAGES
SYSTEMS & STRUCTURES
an international journal
Editor: Bala Kalyanasundaram
the modern tower of Babel
Courtesy of ACM

Computer Languages, Systems & Structures 37 (2011) 168–177

Contents lists available at ScienceDirect

Computer Languages, Systems & Structures

journal homepage: www.elsevier.com/locate/cl

Reducing memory space consumption through dataflow analysis

Ozcan Ozturk *

Computer Engineering Department, Bilkent University, 06800 Bilkent, Ankara, Turkey

ARTICLE INFO

Article history:
Received 22 August 2010
Accepted 18 July 2011
Available online 29 July 2011

Keywords:
Memory
Dataflow analysis
CFG
Compiler

ABSTRACT

Memory is a key parameter in embedded systems since both code complexity of embedded applications and amount of data they process are increasing. While it is true that the memory capacity of embedded systems is continuously increasing, the increases in the application complexity and dataset sizes are far greater. As a consequence, the memory space demand of code and data should be kept minimum. To reduce the memory space consumption of embedded systems, this paper proposes a control flow graph (CFG) based technique. Specifically, it tracks the lifetime of instructions at the basic block level. Based on the CFG analysis, if a basic block is known to be not accessible in the rest of the program execution, the instruction memory space allocated to this basic block is reclaimed. On the other hand, if the memory allocated to this basic block cannot be reclaimed, we try to compress this basic block. This way, it is possible to effectively use the available on-chip memory, thereby satisfying most of instruction/data requests from the on-chip memory. Our experiments with this framework show that it outperforms the previously proposed CFG-based memory reduction approaches.

1. Introduction

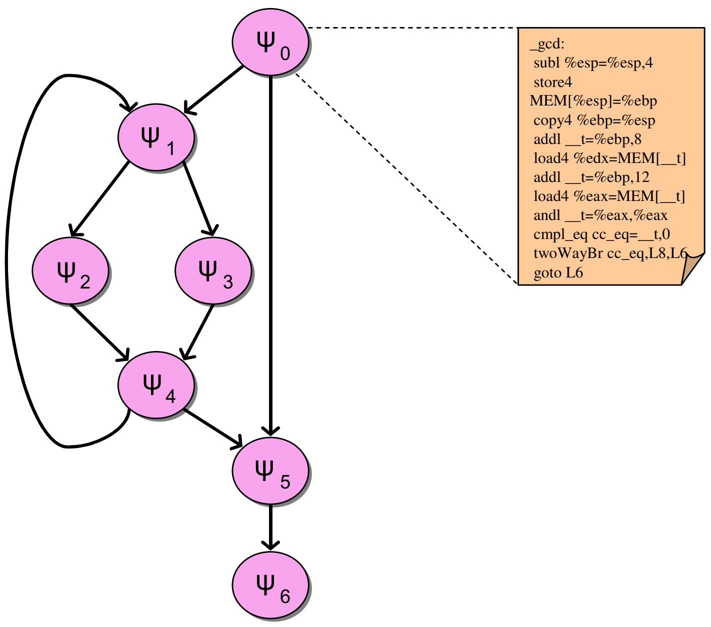
Recent advances in embedded processor design techniques have led to the development of complex embedded systems. Memory is a key parameter in these embedded systems since both code complexity of embedded applications and amount of data they process are increasing. While it is true that the memory capacity of embedded systems is continuously increasing, the increases in the application complexity and dataset sizes are far greater. Moreover, as embedded systems become increasingly complex, there is a growing demand for executing multiple applications concurrently, thereby putting even higher pressure on memory system.

Scratch Pad Memories (SPMs) have received considerable attention as on-chip memory building blocks. This is especially true for embedded systems as SPMs consume less energy and exhibit a very good runtime data locality behavior. Unlike a conventional cache managed by hardware, SPM is controlled by a programmer or a compiler. Therefore, if supported by appropriate compiler analysis and optimizations, SPM can cut the number of off-chip data accesses dramatically. Moreover, when compared to a cache, SPM provides predictability and reproducibility of timings, which is crucial for time critical embedded systems, or other systems where precise timing is important. Prior studies have explored different approaches to exploit the use of SPMs as memory blocks for both instruction and data [2,3,7,9,11,24].

We focus on the efficient use of a two level SPM memory hierarchy shared by multiple applications executing at the same time. In such an SPM memory hierarchy, this paper proposes a control flow graph (CFG) based technique to reduce the memory space consumption of applications. Specifically, it tracks the lifetime of instructions at the basic block level.

* Tel.: +90 312 2903444.
E-mail addresses: ozturk@cs.bilkent.edu.tr, ozturk@nec-labs.com

1477-8424/$ - see front matter © 2011 Elsevier Ltd. All rights reserved.
doi:10.1016/j.cl.2011.07.001



```
_gcd:
 subl %esp=%esp,4
 store4
MEM[%esp]=%ebp
 copy4 %ebp=%esp
 addl __t=%ebp,8
 load4 %edx=MEM[__t]
 addl __t=%ebp,12
 load4 %eax=MEM[__t]
 andl __t=%eax,%eax
 cmpl_eq cc_eq=__t,0
 twoWayBr cc_eq,L8,L6
 goto L6
```

Fig. 1. An example CFG fragment is given for a sample GCD implementation.

Based on the CFG analysis, if a basic block is known to be not accessible in the rest of the program execution, the instruction memory space allocated to this basic block is reclaimed. On the other hand, if the memory allocated to this basic block cannot be reclaimed, our CFG based approach tries to compress the basic block if this basic block is rarely accessed. This way, it is possible to effectively use the available on-chip memory, thereby satisfying 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. In this paper, we make the following contributions:

- The first contribution of this paper is to reclaim the instruction memory that is no longer needed. This is achieved by generating a live set for each basic block in a given application. An example CFG fragment is given for a sample GCD implementation in Fig. 1. If execution reaches basic block Ψ_5, only live basic blocks are Ψ_5 and Ψ_6. Therefore, memory allocated to basic blocks Ψ_0, Ψ_1, Ψ_2, Ψ_3, and Ψ_4 can be deallocated.
- The second contribution of this paper is to extend memory space savings by compressing less frequently accessed basic blocks. Based on the profiling information, compression algorithm selects target basic blocks to compress. One needs to be careful in compressing a basic block as it can cause excessive execution overheads if not carefully selected. For example, if we consider the same CFG given in Fig. 1, one can see that Ψ_0 is followed either by Ψ_1 on the left branch or Ψ_5 on the right branch. If we know based on the profiling information that Ψ_0 is followed by Ψ_5 most of the time, compressing basic blocks Ψ_1, Ψ_2, Ψ_3, and Ψ_4 will reduce the memory space dramatically. As a result, this approach can be expected to be most successful in situations where there exist a few basic blocks with very high reuse.
- Third contribution of this paper is to evaluate the proposed CFG based memory reduction scheme using eleven benchmarks. It also compares our approach to a previously proposed basic block level garbage collection approach. Our experiments with several applications show that our approach can be very useful in increasing the benefits coming from an SPM.
- The last contribution of this paper is to show how saved memory space can be used to increase energy savings in banked memory architectures currently employed in some embedded systems.

The rest of this paper is organized as follows. Section 2 discusses related work, and Section 3 gives the details of the execution environment, dataflow analysis and proposed algorithms. Section 4 presents the results from our experimental evaluation and Section 5 concludes the paper.

2. Related work

Prior SPM studies primarily focused on the data access management. For instance, Panda et al. [16] present a static data partitioning scheme to eliminate the potential conflict misses due to limited associativity of on-chip cache. This approach benefits applications with a number of small (and highly reused) arrays that can fit in the SPM. In [8], authors propose a dynamic SPM management scheme for data accesses. Their framework uses both loop and data transformations to maximize the reuse of data elements stored in the SPM. Cooper et al. [6] show that using the register allocation's coloring paradigm can significantly reduce the amount of memory required for the program. Catthoor [5] discuss how accesses to a software-managed memory hierarchy can be optimized through code/data transformations. In [22], authors present a dynamic allocation method for global and stack data. In the proposed method data to be accessed frequently is copied into the SRAM using compiler-inserted code at fixed points in the program. An integrated hardware/software solution to support SPMs at a high abstraction level is presented in [7]. In contrast to these studies, our work focuses on reducing memory space occupancy of instruction accesses. Consequently, our optimization framework and required compiler analysis are entirely different from theirs.

From an instruction memory angle, work in [4,20] also present SPM management schemes. However, compared to our approach presented in this paper, these schemes are static; that is, the contents of the SPM are decided before the execution and are fixed for the entire execution. While such an approach might be successful with small sized codes, larger codes in general require a dynamic SPM management scheme. Technique presented in [14] tries to reduce the energy consumption of instruction accesses. This scheme is fundamentally different than ours as it relies on hardware (rather than the compiler) to detect opportunities for exploiting the SPM. As mentioned earlier in the paper, such an approach is costly and requires significant modifications to hardware. An optimal SPM mapping approach has been proposed in [3] for code segments that works directly on application binaries. Dynamic programming algorithm is applied to solve the mapping problem. In [21], authors present a compiler-managed instruction store architecture (K-store) that places the computation intensive loops in an SPM like SRAM memory and allocates the remaining instructions to a regular instruction cache. Execution is switched (at runtime) dynamically between the instructions in the traditional instruction cache and the ones in the K-store, by inserting jump instructions. Approach presented in [24] use the SPM for storing instructions and propose a generic cache aware SPM allocation. Udayakumaran et al. [23] propose a dynamic strategy for embedded systems with SPMs, where data that is about to be accessed frequently is copied into the scratch pad using compiler-inserted code at fixed and infrequent points in the program. Verma et al. [25] present an allocation technique to insert instructions which dynamically copy code segments and variables on the SPM. In [18], authors use compiler managed dynamic placement algorithm to keep hot code segments in SPM. Li et al. [10] propose a compiler directed memory coloring approach to assign static data onto SPM. Werth et al. [26] propose an approach to automatically fragment and load code into local SPM. Basic block level management of an instruction memory is proposed in [15]. Our work is different as we employ basic block compression for the basic blocks that are not known to be dead. However, BBGC-base keeps them as they are, whereas BBGC-aggressive deallocates them. While the latter reduces the memory space drastically, it incurs performance penalty due to the false speculation. Our approach generates better results with minimal overheads as will be shown in the experimental results.

There also exist several cache memory-based studies that aim at improving the performance of instruction caches through compiler-directed modifications. Works presented in [17] and [12] present algorithms to bring instructions (basic blocks) with temporal affinity together in the address space in an attempt to improve instruction cache hit rates. In [13], authors also consider procedure merging to further increase the instruction cache performance. Work in [19] presents trace cache, a special instruction cache that captures dynamic instruction sequences. In contrast to all these cache-based studies, our work focuses on software-managed instruction memories, and demonstrates how dead basic blocks can be collected for reducing average memory occupancy.

3. Basic block memory reduction

3.1. *Target architecture*

In this paper, we focus on a single CPU-based embedded architecture with an SPM, of which the high level diagram is shown in Fig. 2. This architecture contains a two-level software-managed memory hierarchy where on-chip SPM space is divided into instruction memory and data memory portions. Moreover, the SPM space is shared by all applications running concurrently. In addition to the SPM space, there is also a large off-chip memory space. While the architecture contains both data and instruction memories, in this work we exclusively focus on instruction accesses. Multicore processors have rapidly gained traction in the embedded domain and will soon dominate the processors on many of these systems. In such architectures, each processor core can potentially have its own SPM or there can even be a shared SPM space. Therefore, other factors need to be considered in a multicore environment. We concentrate mainly on the management of the instruction memory and the number of processor cores to use for a given application is orthogonal to the focus of this paper. Our goal in this paper is rather to study the management of an instruction memory space of a processor.

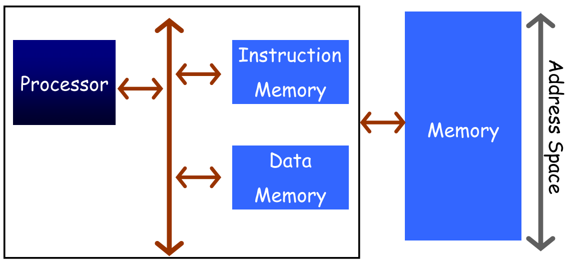


Fig. 2. Two-level software-managed memory architecture.

3.2. Our approach

Our approach tries to deallocate the instruction memory that is no longer needed. We achieve this by deallocating the memory space of the dead basic blocks at every point of execution. Our approach is similar to the one presented in [15] where lifetime of instructions are tracked at the basic block granularity. In their approach, there are two versions of the proposed scheme, BBGC (basic block-level garbage collection) and BBGC-aggressive. In BBGC-aggressive, memory allocated for a basic block is deallocated even if the basic block is not dead, but its next access is predicted to be too far in the execution. Potentially, this increases the execution time when the basic block in question needs to be brought into the on-chip memory. Alternatively, our approach tries to compress the basic blocks that are predicted to be too far in the execution, instead of sending them off-chip. This reduces the memory allocated for such basic blocks, simultaneously keeping them still in the on-chip memory for later accesses.

Instruction memory effectiveness can be increased by making use of basic block compression since, at any given time, not all the basic blocks in the instruction memory have the same criticality. While some of them might be in active use, others will not be accessed until some time in the future. Therefore, it might be possible to keep the latter type of basic blocks in a compressed form to increase the available space for new applications.

Our specific compression algorithm selects the basic blocks to be compressed by looking at their predicted next uses. One needs to be careful in compressing a basic block as it can cause excessive execution overheads if not carefully selected. This approach can be expected to be most successful in situations where there exist a few basic blocks with very high reuse. Consider the example CFG given in Fig. 3, where frequencies and number of times each CFG edge is taken are given next to the edges. As can be seen from this figure, Ψ_0 is dead when execution reaches Ψ_1. On the other hand, basic blocks Ψ_1 and Ψ_2 are accessed over and over for almost a million times. Rest of the basic blocks, Ψ_3 through Ψ_{k-1} shown with "*Remaining BBs*", are not used for a long time and can be compressed to save on-chip memory space. As a result, this approach can be expected to be most successful in situations where there exist a few basic blocks with very high reuse. By doing so, we greatly reduce the memory space with minimal execution overheads.

As stated earlier, the drawback of compression is that if such a (compressed) basic block needs to be accessed later, it first needs to be decompressed, which typically takes time and consumes energy. Because of this, the number of decompressions should be minimized as much as possible. In fact, for such a strategy to be successful, the cost of decompressing a basic block within the SPM must be lower than accessing it (in an uncompressed form) from the off-chip memory. To limit the number of decompressions, we do not compress the basic blocks that are part of a cyclic path in the CFG. This way it is possible to eliminate the basic blocks that are being accessed over and over within a cyclic path.

3.3. Algorithm

One needs to make several important decisions to manage the basic blocks of a given application: (1) Which basic blocks should be declared as dead? (2) If a basic block is found to be alive, should that basic block be compressed, or be left uncompressed? (3) Where should a newly created basic block be stored in the SPM? (4) When the current use of a data block is over, should we compress it or not? In this section, we present an algorithm to capture compiler-based SPM management heuristic based on data basic block analysis.

Our algorithm that demonstrates how SPM management heuristic operates is shown in Fig. 4. The algorithm iterates over all the basic blocks in the application code. In this algorithm, $\mathcal{P}_i$, $\mathcal{S}_i$, and $\mathcal{D}_i$ correspond to the set of predecessors, the set of successors, and the set of dead basic blocks, respectively. These sets are generated offline by the compiler and

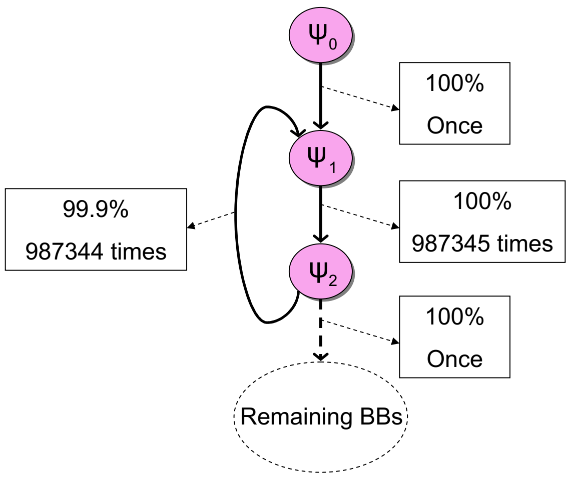


Fig. 3. An example CFG fragment where a few basic blocks are highly reused.

```
$\mathcal{N}$: – number of basic blocks in the procedure;
$\mathcal{T}_c$: – reuse threshold to compress a basic block;
$\mathcal{T}_d$: – reuse threshold to decompress a basic block;
$\mathcal{L}$: – the set of currently live basic blocks;
$\mathcal{C}$: – the set of compressed basic blocks;
$\mathcal{P}_i$: – the set of predecessors of basic block $i$;
$\mathcal{S}_i$: – the set of successors of basic block $i$;
$\mathcal{D}_i$: – the set of dead basic blocks $i$;
$l$: – the basic block being executed;

  $\mathcal{L}$ =all the basic blocks in the procedure
  $\mathcal{C} = \emptyset$
  for $i = 1$ to $\mathcal{N}$ in reverse postorder{
    generate $\mathcal{P}_i$ and $\mathcal{S}_i$
    generate $\mathcal{D}_i$
  }

  procedure execute($l$) {
    if ($l \in \mathcal{C}$) { // l is compressed
      $l.r = 0$; // the next use time of $l$ has been mispredicted
      decompress($l$);
      select the memory bank with $max_{aff}(l)$;
      // memory bank with available space
      $\mathcal{C} = \mathcal{C} - \{l\}$;
    }
    // l is already decompressed
// deallocate the memory of the dead basic blocks
    for all $t \in \mathcal{D}_l$ {
      evict($t$); //deallocate the memory
      if ($t \in \mathcal{C}$) // t is compressed
        $\mathcal{C} = \mathcal{C} - \{t\}$;
      $\mathcal{L} = \mathcal{L} - \{t\}$;
    }
// compress the basic blocks with long reuse distance
    for each $t \in \mathcal{P}_l$ such that $t.r \geq \mathcal{T}_c$ {
      compress($t$);
      select the memory bank with $max_{aff}(t)$;
      // memory bank with available space
      $\mathcal{C} = \mathcal{C} + \{t\}$;
    }
// decompress the basic blocks to be reused soon
    for each $t \in \mathcal{S}_l$ such that $t.r < \mathcal{T}_d$ {
      decompress($t$);
      select the memory bank with $max_{aff}(t)$;
      // memory bank with available space
      $\mathcal{C} = \mathcal{C} - \{t\}$;
    }
  }
```

Fig. 4. Our algorithm that demonstrates how SPM management heuristic based on basic block analysis operates.

populated to our tool. Similarly, we keep track of the live basic blocks and compressed basic blocks using the $\mathcal{L}$ set and $\mathcal{C}$ set, respectively. Note that, uncompressed basic blocks can easily be obtained by $\mathcal{L}-\mathcal{C}$.

Although, the next basic block to be executed is found to be decompressed in most cases. Rarely, it may need to be decompressed due to the mispredicted reuse distance. Once it is decompressed, compressed set $\mathcal{C}$ is updated. Next, memory allocated to the basic blocks that are marked as dead ($\mathcal{D}_l$) is deallocated. This is performed in the second for loop by using the list of dead basic blocks when the execution reaches basic block *l*. Both compressed set $\mathcal{C}$ and live set $\mathcal{L}$ are updated.

In the last part, algorithm identifies the basic blocks among the predecessors ($\mathcal{P}_l$) to compress soon. These basic blocks are selected based on their reuse distances, *t.r*. These values are obtained for each basic block pair through profiling. Similarly, basic blocks that are going to be accessed soon (among the successors, $\mathcal{S}_l$) are decompressed. We use two different threshold values to decide, namely, $\mathcal{T}_c$ and $\mathcal{T}_d$ for compressions and decompressions, respectively. We initially tested our approach using a single threshold for both compression and decompression. However, due to different costs for compression and decompression, we observed that using two different thresholds gives better results.

3.4. Energy reduction in multi-bank memories

We now discuss how our approach can be used for reducing energy consumption of a multi-bank on-chip memory. The benefits come from the fact that our basic block placement strategy stores the blocks with similar lifetimes consecutively in memory. Therefore, they are likely to reside in the same memory bank, and if all the blocks stored in a given bank are dead, the bank can be placed into a low-power mode to save energy. In Fig. 4, this is achieved by selecting the memory bank using the call max_{aff}. It selects the memory bank with maximum affinity by looking at the resident basic blocks. This is the case for both compressions and decompressions. We compare this approach with a sequential layout, where basic blocks are distributed based on the CFG reverse postorder. Our results indicate that the effectiveness of this strategy increases when the number of banks is increased (under the same total on-chip memory capacity).

4. Experimental evaluation

4.1. Simulation platform and implementation

Trimaran is a compiler/simulator infrastructure that provides a vehicle for implementation and experimentation in state-of-the-art research in compiler techniques for instruction level parallelism (ILP) [1]. In the Trimaran environment, a program flows through Impact, Elcor, and the cycle-level simulator. Impact applies machine-independent classical optimizations and transformations to the source program, whereas Elcor is responsible for machine-dependent optimizations and scheduling (see Fig. 6). We modified the Elcor part to insert special instructions for bringing procedures to the on-chip memory compressing/decompressing and deleting the dead basic blocks when possible. The increase in compilation time due to our algorithms was around 20% on the average, and the increase in code size (due to the extra instructions inserted by the compiler) was less than 3% for all the applications we tested. The cycle-level simulator was modified to track the status of each basic block that resides in the on-chip memory. It must be emphasized that while we model a VLIW architecture, our approach is certainly applicable to other style of architectures as well. Essentially, it is a memory space optimization technique that could be used with different types of embedded processors.

4.2. Results

We used eleven benchmarks to test the effectiveness of our scheme. Table 1 gives these benchmarks and their salient features. The third column gives the number of basic blocks in the CFGs of each benchmark, and the fourth column gives the number of basic block visits (executions) at runtime. The next column shows the number of function calls in each application at runtime, and the sixth column gives the maximum memory space occupied by the basic blocks of the applications (at runtime). Note that, if no memory space optimization is performed, this is the "memory occupancy" (or "memory consumption") of the application. Our objective is to reduce the memory occupancy over the course of execution by exploiting the lifetimes of basic blocks. The last column of Table 1 gives the number of execution cycles for each benchmark, when no space optimization is used.

The graphs in Fig. 5 show the memory occupancy of the benchmarks for the original case (marked "Original"), the case using a previously proposed scheme [15] (marked "BBGC"), and the case using our approach (marked "Our Approach"). Note that, the *x*-axis represents the time divided into slices of equal size. Comparing our scheme with BBGC version (marked "BBGC"), one can see that our scheme saves more memory space (except in benchmarks `apsi`, `cordic`, and

Table 1
The characteristics of the benchmark codes used in this study.

Benchmark	Source	Number of basic blocks	Number of basic block visits	Number of function calls	Code size (KB)	Number of execution cycles
`adi`	Livermore	17	70,605	1	414	283,111
`apsi`	Perfect Club	25	50,411	14	695	144,923
`bmcm`	Perfect Club	25	1,060,602	1	413	7,255,040
`compress`	Spec	308	5,543,178	34,422	1799	20,846,970
`cordic`	MediaBench	54	300,069	1	1234	2,105,869
`eflux`	Perfect Club	43	69,770	1	471	555,293
`g721encode`	MediaBench	357	63,815,783	1,475,191	1766	165,938,959
`mxm`	Spec	17	1,060,605	1	343	7,182,875
`rawcaudio`	MediaBench	98	2,495,142	228	982	7,434,169
`tomcatv`	Spec	51	61,684	1	697	552,610
`tsf`	Perfect Club	38	324,762	16	469	2,543,920

g721encode where the two schemes generate similar results). We see that different benchmarks exhibit generally different memory occupancy trends when BBGC or our approach is used. However, in all the benchmarks, our approach reduces the memory occupancy further down compared to BBGC. Fig. 7 presents the savings in the memory occupancy, *averaged* over the execution time. We see that BBGC and BBGC-aggressive reduces the average instruction memory occupancy by 35% and 43%, respectively, whereas our approach saves 17% more memory space as compared to the BBGC and 9% more memory space as compared to the BBGC-aggressive. The savings are much higher with benchmarks such as bmcm and compress, whereas apsi and rawcaudio show relatively smaller benefits. This can be explained as follows. In both bmcm and compress, there is a loop that consumes most of the execution time ($\sim 70\%$) and this loop comes close to the end of the program execution. Consequently, when the execution reaches this loop and iterates in it, most of the basic blocks encountered earlier in the execution are already dead; that is, their memory space has already been deallocated. In contrast, in apsi and rawcaudio, the execution mostly iterates in a loop that is at the very beginning of the CFG. Therefore, for a long time, the memory space that holds other basic blocks in the program cannot be reclaimed.

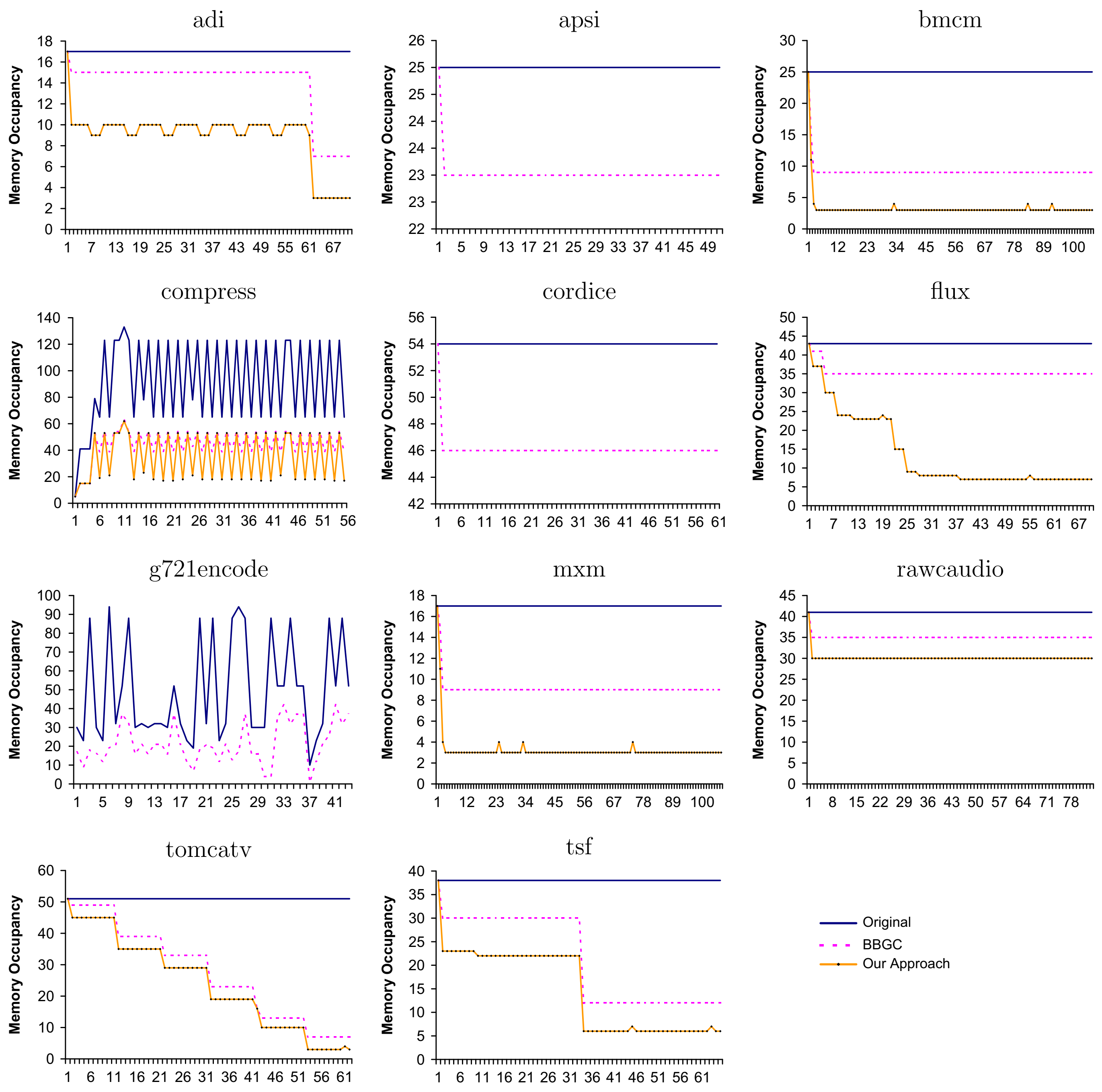


Fig. 5. Memory occupancies of the benchmark codes in our suite (note that the curves for "BBGC" and "Our Approach" are the same for apsi, cordic, and g721encode).

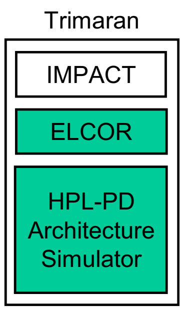


Fig. 6. Major components of Trimaran. Shaded portions are modified to implement our approach.

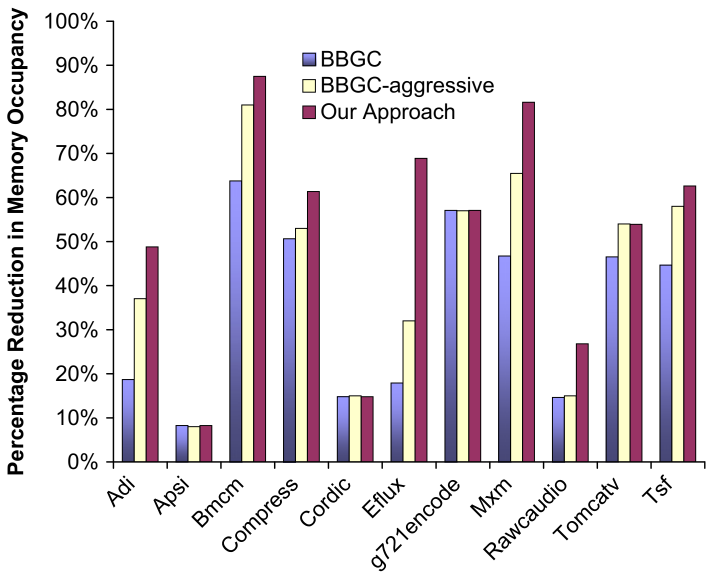


Fig. 7. Average improvements in memory occupancy of basic blocks.

Table 2
Performance overheads incurred in the proposed approaches.

Benchmark	**Performance overheads for BBGC** (%)	**Performance overheads for our approach** (%)
adi	2.67	3.14
apsi	3.59	4.27
bmcm	1.53	2.55
compress	2.78	8.16
cordic	4.40	5.30
eflux	1.36	4.69
g721encode	3.86	7.90
mxm	2.01	3.06
rawcaudio	3.41	2.66
tomcatv	1.65	2.81
tsf	3.00	4.15
Average	2.75	4.43

However, as has been discussed earlier, one drawback of this scheme is the increase in the execution cycles due to compressions and decompressions. Assuming 1 cycle on-chip memory latency, 20 MB/s LZO Compression/Decompression Rate, and 80 cycle off-chip memory latency, Table 2 gives the percentage increase in execution cycles due to BBGC and our approach. The increases are with respect to the execution cycles given in the last column of Table 1. We see that the average increases brought by the BBGC and our scheme are 2.75% and 4.43%, respectively. The percentages in this table include *all* the performance overheads incurred by these schemes. The reason for the relatively large increase in execution cycles of g721encode and compress when our scheme is used is the large number of decompressions due to the basic blocks that have been compressed not long ago. This is mainly caused by loops with small execution times. While this

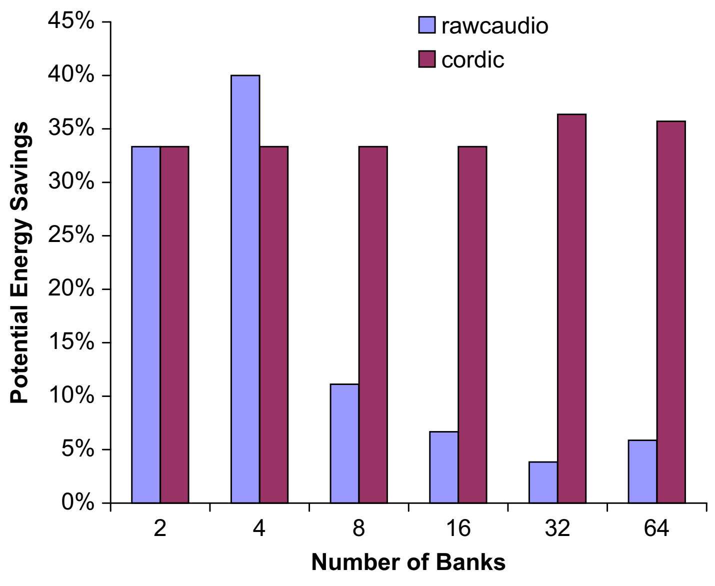


Fig. 8. Average number of bank savings brought by our placement strategy over the sequential layout.

drawback can be fixed in part by stipulating that a block is deallocated only if there are $k>1$ loops between its successive visits, we did not pursue this direction further.

We next evaluate the influence of our approach to energy savings. The scheme evaluated is the one discussed in Section 3.4. As we have discussed earlier, the saved memory space can also be used for energy savings in a banked memory architecture. The potential benefits of pursuing this direction are shown in Fig. 8 for `rawcaudio` and `cordic`. To obtain these results, the memory is scanned at each basic block access at runtime, and considering the number of banks (x-axis), we counted what percentage of the banks would hold only dead blocks, and thus would be put in a low-power mode. The savings are then given as fractions of the savings obtained using sequential layout. Note that, while our absolute savings actually increase with the increased number of banks, the results given in this graph do not reflect this trend as they are normalized with respect to the sequential layout. It must be mentioned that these are just potential benefits, as managing low-power modes would typically involve some power/performance overheads, that also need to be accounted for. However, these graphs still indicate that large energy gains might be possible through our approach.

5. Conclusion

This paper shows how CFG analysis can be used for reducing the memory space consumption. Specifically, instruction memory requirements of a given application is reduced as much as possible by tracking the lifetime of basic blocks. First, we deallocate the memory space allocated to the basic blocks that are no longer accessible. Second, we compress the basic blocks that are alive but rarely accessed. The compiler provides the CFG and the corresponding access frequencies of CFG edges to our framework, which in turn determines the deallocation and compression decisions. This paper also presents experimental evidence, demonstrating the impact of the proposed approach in practice. The results indicate that the CFG-based techniques presented in this paper is very effective in reducing memory space consumption, and our savings are consistent across a range of values of major simulation parameters.

Acknowledgments

This research is supported in part by TUBITAK Grant 108E233, by a Grant from IBM, and by a Marie Curie International Reintegration Grant within the Seventh European Community Framework Programme.

References

[1] The Trimaran Compiler Infrastructure 〈http://www.trimaran.org〉.
[2] Angiolini F, Benini L, Caprara A. Polynomial-time algorithm for on-chip scratchpad memory partitioning. In: Proceedings of the 2003 international conference on compilers, architecture and synthesis for embedded systems; 2003. p. 318–26.
[3] Angiolini F, Menichelli F, Ferrero A, Benini L, Olivieri M. A post-compiler approach to scratchpad mapping of code. In: CASES '04: proceedings of the 2004 international conference on compilers, architecture, and synthesis for embedded systems; 2004. p. 259–67.
[4] Bellas N, Hajj I, Polychronopoulos C. A new scheme for i-cache reduction in high performance processors. In: Proceedings of the power driven micro-architecture workshop; 1998.
[5] Catthoor F, Danckaert K, Kulkarni C, Brockmeyer E, Kjeldsberg P, Achteren TV, Omnes T. Data access and storage management for embedded programmable processors. Boston, MA, USA: Kluwer Academic Publishers; 2002.

[6] Cooper KD, Harvey TJ. Compiler-controlled memory. In: ASPLOS-VIII: proceedings of the eighth international conference on architectural support for programming languages and operating systems; 1998. p. 2–11.
[7] Francesco P, Marchal P, Atienza D, Benini L, Catthoor F, Mendias JM. An integrated hardware/software approach for run-time scratchpad management. In: Proceedings of the 41st annual conference on design automation; 2004. p. 238–43.
[8] Kandemir M, Ramanujam J, Choudhary A. Exploiting shared scratch pad memory space in embedded multiprocessor systems. In: DAC '02: proceedings of the 39th conference on design automation; 2002. p. 219–24.
[9] Kandemir M, Ramanujam J, Irwin J, Vijaykrishnan N, Kadayif I, Parikh A. Dynamic management of scratch-pad memory space. In: Proceedings of the 38th conference on design automation; 2001. p. 690–5.
[10] Li L, Feng H, Xue J. Compiler-directed scratchpad memory management via graph coloring. ACM Trans Archit Code Optim 2009;6(3):1–17.
[11] Marwedel P, Wehmeyer L, Verma M, Steinke S, Helmig U. Fast, predictable and low energy memory references through architecture-aware compilation. In: ASP-DAC '04: proceedings of the 2004 conference on Asia South Pacific design automation; 2004. p. 4–11.
[12] McFarling S. Program optimization for instruction caches. SIGARCH Comput Archit News 1989;17(2):183–91.
[13] McFarling S. Procedure merging with instruction caches. SIGPLAN Not 1991;26(6):71–9.
[14] Motorola. M-core—mmc2001 reference manual, Motorola corporation; 1998.
[15] Ozturk O, Kandemir M, Irwin MJ. Bb-gc: basic-block level garbage collection. In: DATE '05: proceedings of the conference on design, automation and test in Europe; 2005.
[16] Panda PR. Memory bank customization and assignment in behavioral synthesis. In: ICCAD '99: proceedings of the 1999 IEEE/ACM international conference on computer-aided design; 1999. p. 477–81.
[17] Pettis K, Hansen RC. Profile guided code positioning. SIGPLAN Not 1990;25(6):16–27.
[18] Ravindran RA, Nagarkar PD, Dasika GS, Marsman ED, Senger RM, Mahlke SA, Brown RB. Compiler managed dynamic instruction placement in a low-power code cache. In: CGO '05: proceedings of the international symposium on code generation and optimization; 2005. p. 179–90.
[19] Rotenberg E, Rotenberg E, Bennett S, Bennett S, Smith JE, Smith JE. Trace cache: a low latency approach to high bandwidth instruction fetching. In: Proceedings of the 29th international symposium on microarchitecture; 1996. p. 24–34.
[20] Steinke S, Wehmeyer L, Lee B, Marwedel P. Assigning program and data objects to scratchpad for energy reduction. In: Proceedings of the conference on design, automation and test in Europe; 2002.
[21] Suresh DC, Najjar WA, Yang J. Power efficient instruction caches for embedded systems. In: Proceedings the fifth international workshop on embedded computer systems: architectures, modeling, and simulation; 2005.
[22] Udayakumaran S, Barua R. Compiler-decided dynamic memory allocation for scratch-pad based embedded systems. In: CASES '03: proceedings of the 2003 international conference on compilers, architecture and synthesis for embedded systems; 2003. p. 276–86.
[23] Udayakumaran S, Dominguez A, Barua R. Dynamic allocation for scratch-pad memory using compile-time decisions. ACM Trans Embed Comput Syst 2006;5(2):472–511.
[24] Verma M, Wehmeyer L, Marwedel P. Cache-aware scratchpad allocation algorithm. In: Proceedings of the conference on design, automation and test in Europe; 2004.
[25] Verma M, Wehmeyer L, Marwedel P. Dynamic overlay of scratchpad memory for energy minimization. In: CODES+ISSS '04: proceedings of the second IEEE/ACM/IFIP international conference on hardware/software codesign and system synthesis; 2004. p. 104–9.
[26] Werth T, Flossmann T, Klemm M, Schell D, Weigand U, Philippsen M. Dynamic code footprint optimization for the ibm cell broadband engine. In: IWMSE '09: proceedings of the 2009 ICSE workshop on multicore software engineering; 2009. p. 64–72.