DRIM : A Low Power Dynamically Reconfigurable Instruction Memory Hierarchy for Embedded Systems

Zhiguo Ge, Weng-Fai Wong
Department of Computer Science,
National University of Singapore
{gezhiguo,wongwf}@comp.nus.edu.sg

Hock-Beng Lim
ST Engineering
limhb@steng.com.sg

Abstract

Power consumption is of crucial importance to embedded systems. In such systems, the instruction memory hierarchy consumes a large portion of the total energy consumption. A well designed instruction memory hierarchy can greatly decrease the energy consumption and increase performance. The performance of the instruction memory hierarchy is largely determined by the specific application. Different applications achieve better energy-performance with different configurations of the instruction memory hierarchy. Moreover, applications often exhibit different phases during execution, each exacting different demands on the processor and in particular the instruction memory hierarchy. For a given hardware resource budget, an even better energy-performance may be achievable if the memory hierarchy can be reconfigured before each of these phases. In this paper, we propose a new dynamically reconfigurable instruction memory hierarchy to take advantage of these two characteristics so as to achieve significant energy-performance improvement. Our proposed instruction memory hierarchy, which we called DRIM, consists of four banks of on-chip instruction buffers. Each of these can be configured to function as a cache or as a scratchpad memory (SPM) according to the needs of an application and its execution phases. Our experimental results using six benchmarks from the MediaBench and the MiBench suites show that DRIM can achieve significant energy reduction.

1 Introduction

With the proliferation of portable devices such as mobile phones, digital cameras, etc, power consumption has become a major design consideration. As the instructions are fetched almost every cycle, the instruction delivery system constitutes a significant portion of the total energy consumption by the processor. Power consumption affects the battery life and the heat dissipation of portable devices, which in turns affects their usability. Thus, designing an energy efficient instruction delivery system is very important for embedded systems.

Several approaches have been proposed for the reduction of energy consumption in caches. First, there were proposals for customizable and reconfigurable caches that adapt to the characteristics of a specific application. Cache banks are shut down and the cache associativity is reconfigured when necessary in order to decrease energy consumption [18, 1].

Another popular method is the use of scratchpad memories (SPMs) as energy efficient on-chip buffers [2]. The SPM is more energy efficient than a cache since it does not require tag storage and its control logic is simpler. Tag access and comparison occur with every cache access and therefore consume significant amount of energy. Furthermore, a SPM can be utilized by the application in such a way that instruction conflicts are reduced. Mapping frequently used data and instructions to pure SPM or hybrid SPM and cache architectures have also been explored [16, 9]. However, in these studies, the memory hierarchies were assumed to be fixed.

Other researchers [11, 15] performed design space explorations for the on-chip SPM, utilizing the characteristics of an given application to further improve the energy savings. There are two issues with such approaches. Firstly, the design choices for the memory hierarchy may be limited in reality. Secondly, these approaches failed to take advantage of the phased behavior of applications during their execution.

In this paper, we propose a novel dynamic reconfigurable instruction memory hierarchy (DRIM) for embedded systems. Our proposed architecture consists of four banks of storage, each of which can be dynamically reconfigured to be part of a cache or a SPM to suit an application and its execution phases. We will also describe an algorithm that supports the dynamic reconfiguration of DRIM and the selection and allocation of code to be executed from the scratchpad memory. Our experimental results using six benchmarks from the Mediabench and the MiBench suites show that our framework can achieve significant energy savings.

The rest of the paper is organized as follows. In Section 2, we will discuss related works and our contributions. Section 3 introduces our DRIM architecture. We then present the compiler framework and algorithms for partitioning the memory hierarchy reconfiguration and allocating instruction to the SPM in Section 4. In Section 5, we present our experimental

methodology and discuss the results. We conclude this paper in Section 6.

2 Related Work

Several researchers have studied the use of SPM in the instruction memory hierarchy, with the aim of saving energy in embedded systems. Instruction may be statically mapped into a given instruction memory hierarchy consisting of only of SPM or a mixture of cache and SPM [16, 9]. Algorithms to statically partition instructions for a SPM of a given size (with or without the presence of a cache) have been proposed. However, these works do not consider dynamic instruction replacement and the possibility of changing hardware configurations. Such schemes suffer from a lack of flexibility and the SPM is not efficiently used.

Dynamic instruction replacement to improve the utilization of SPM has also been studied [7, 4, 14]. Different instruction blocks may occupy and reuse the same SPM entries to improve the SPM's efficiency. Significantly greater energy savings can be achieved over the static methods. However, the above works did not consider tuning the architecture parameters for different applications. Several other researchers studied the problem of design space exploration so as to find the best memory hierarchy parameters for a given application. Ge et. al [11] partitioned the given storage resource budget into a SPM and a cache, according to the application's characteristics. Van der Aa, et. al. [15] performed the exploration for optimal configurations of the instruction loop buffer given an application, and mapped selected loops into these loop buffers so as to reduce the energy consumption.

The existing work on instruction SPM can be classified into three categories: (i) static architecture with static mapping, (ii) static architecture with some dynamic replacement strategies, and (iii) static architecture exploration with static mapping. None of these considered the dynamic tuning architectural parameters. Kondo et. al. [6] proposed a dynamic reconfigurable data memory hierarchy consisting of SPM and cache. However, they did not consider the instruction memory hierarchy. The main contributions of our work are as follows:

1. We propose a new dynamic reconfigurable instruction memory hierarchy (DRIM), which enables more flexible use of a SPM than previous methods. It can be reconfigured for different applications instead of being tuned just for a particular program. Furthermore, DRIM can be reconfigured for the different phases of execution of an application, so as to minimize the energy consumption of each phase.

2. We developed a compilation strategy to support this reconfiguration memory hierarchy.

To the best of our knowledge, dynamic reconfiguration of SPM for instruction memory hierarchy has yet to be studied.

3 DRIM Architecture

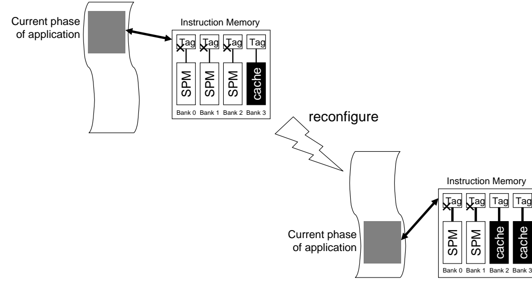

Figure 1: Reconfiguring memory at runtime.

Differences between applications as well as between phases of execution within an application can best be exploited if the memory hierarchy can be reconfigured. Figure 1 illustrates the idea of how the DRIM architecture works. In the configuration of DRIM that we designed and studied, we reconfigured four banks of storage dynamically as cache or SPM.

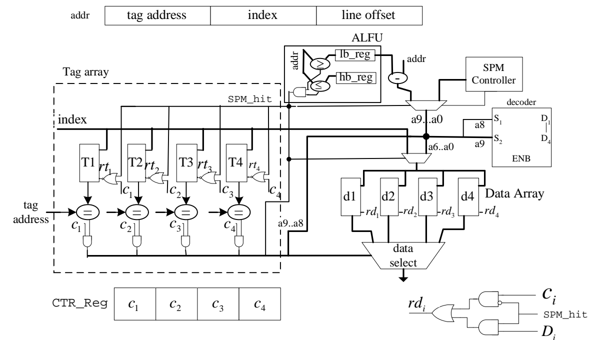

Figure 2: DRIM architecture.

The architecture of DRIM is shown in Figure 2. It consists of the tag logic, the data array, the SPM control logic, and other logic. The DRIM architecture is based on a four way associative cache architecture. One important difference is that tag and data access is controlled by the SPM control logic and a control register known as `CTR_Reg`. In Figure 2, the `CTR_Reg` is collectively the four bits, c_1, to c_4. These four bits determine the configuration of DRIM. Each bit is associated with one bank of tag and data storage. If the bit is one, the corresponding data bank will be configured as a SPM. Also, the tag bank will be gated, thereby decreasing its activity, which in turns results in energy savings. The value of the `CTR_Reg` is manipulated by the processor.

The *address lookup functional unit* (ALFU) determines whether an instruction is residing in the SPM or not. It consists of two address registers and two parallel comparators. The two registers, `ub_reg` and `lb_reg`, hold the upperbound and the lowerbound addresses for the instruction block

that is to reside in the SPM, respectively. If the address of an instruction to be fetched falls within the range of these two registers, then it is in the SPM and the ALFU will generate the `SPM_hit` signal. This signal controls the selection and gating of the tag and the data banks. In the DRIM design presented here, there is only one pair of `ub_reg` and `lb_reg`. As a result, only one block of instructions may reside in the SPM banks at any one time.

The SPM controller performs the loading of instructions from main memory into the SPM, as well as the updating of the upperbound and lowerbound registers.

The tag and data banks are selected or gated according to the value of `CTR_Reg`. The address of the instructions in SPM will determine which SPM bank should be accessed. For each tag array t_i, the gate signal is rt_i:

$$rt_i = \sim\texttt{SPM_hit} \wedge \sim c_i = \sim(\texttt{SPM_hit} \vee c_i)$$

In other words, all the tag banks will be gated and de-activated if `SPM_hit` is asserted. A de-asserted `SPM_hit` implies that the instruction to be fetched is not in the SPM. In that case, only the tags corresponding to the banks configured as cache, i.e. those whose c_i is true, will be searched.

For data array i, the corresponding gate signal, rd_i is:

$$rd_i = (\sim\texttt{SPM_hit} \wedge \sim c_i) \vee (\texttt{SPM_hit} \wedge D_i)$$

where D_i is the data bank selection signal. If an instruction is not in SPM, i.e. `SPM_hit` is false, the data array of the storage banks configured as cache will be accessed. Otherwise, i.e. if `SPM_hit` is true, the SPM bank containing the instruction will be selected by D_i. The following simple example illustrates how D_i can be computed: Suppose all four data banks are configured as SPM and the size of each data bank is 256 bytes. In this case, bank 1, 2, 3, and 4 will hold instructions for which the last 10 bits of the addresses are in the range 0x000 to 0x0FF, 0x100 to 0x1FF, 0x200 to 0x2FF, and 0x300 to 0x3FF, respectively. Clearly, the two most significant bits can then be used as the bank selection signal D_i. The remaining eight bits can be used as the address supplied to the data banks.

4 Compiler Framework

DRIM requires the compiler's support to realise its dynamic reconfiguration. The compiler also has to insert instructions into an application in order to dynamically load selected instructions into the SPM. We have developed a compiler framework that performs these functions.

4.1 Compilation Flow

The structure of our compilation flow is shown in Figure 3. The inputs are the given application and the storage resource budget for the instruction memory hierarchy. The outputs are the partitioning decision for the instruction memory hierarchy custom-made for the application, and the transformed application with an optimized instruction layout. The framework consists of several steps:

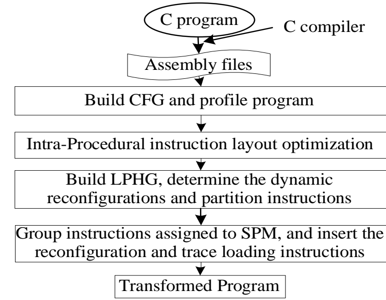

Figure 3: Design Flow

- *Profiling the application*: First, profiling is used to obtain the runtime characteristics of the application. The information collected include the execution counts of the edges of the control flow graphs (CFGs) of all the procedures and the number of the procedure invocations. This is done by building a CFG for each procedure, and then adding instructions to instrument each basic block of a CFG. The instrumented program is executed to get the required execution statistics.

- *Intra-procedural instruction layout optimization*: The goal of this step is to optimize the instruction layout within each procedure according to the profiling statistics obtained in the previous step. We used the Top-down Positioning algorithm proposed by Pettis and Hansen [13] to perform intra-procedural layout optimization. This step brings the frequently executed basic blocks together to make it easier to extract a frequently executed trace.

- *Determining the reconfiguration and partitioning instructions to SPM*: In this step, the application's runtime profile is analyzed so as to determine the suitable points to reconfigure DRIM and the corresponding configurations. At the same time, the instructions blocks are partitioned to the dynamically configured SPM banks.

- *Grouping partitioned instruction blocks, and inserting reconfiguration and trace load instructions*: After the preceding step, the architectural configurations for different phases are determined and the instructions are partitioned to SPM banks. At this step, we generate code chunks namely traces by taking out and grouping the instruction blocks assigned to SPM. Then, the instructions for architecture reconfiguration and trace loading are inserted into the application. All the instructions in an trace are contiguous and the whole trace will be loaded into

SPM when a loading happens. The jump instructions might need to be added to maintain the control flow relations between basic blocks.

We evaluate the proposed framework using the Simplescalar tool set [3]. The Simplescalar simulator was extended to support DRIM. We also built an instruction optimization tool which performs the program profiling and the intra-procedural instruction layout optimization.

4.2 Dynamic reconfigurations and instruction replacement

This section describes the second innovation of this paper other than the DRIM architecture, namely an algorithm to decide where and when to reconfigure DRIM as well as deciding which instructions should go into the SPM. The reconfiguration and the instruction allocation are determined by the phaseal behavior of the execution of an application. Our proposed algorithm is shown in Algorithm 1. The algorithm uses the Loop-Procedure Hierarchy Graph (LPHG) [10] to represent a program. The LPHG captures all the loops, and procedure calls of an application as well as their relations. In order to estimate the cache misses for loops, the sizes of loops in LPHG are computed (line 2 of Algorithm 1).

We assume that most of the energy consumed by instruction fetching as well as most of the instruction cache conflicts occurs inside loops. The intuition is that if the number of loop iteration are large enough to outweigh the overhead of the reconfiguration and trace loading, then the loop should be placed into the SPM. If the loop is too big to fit into the SPM, then the cache is used to buffer the rest of it.

In a LPHG, the deeper a loop is, the higher is its execution frequency. The algorithm therefore starts from the leaf loops and work toward their parent loops. If the number of the loop iterations is larger than a threshold value, the energy savings obtained from the usage of SPM will overweigh the overhead of reconfiguring DRIM. It is then beneficial to reconfigure the data storage banks into SPM and use it. For this paper, we empirically set the threshold value to be 30.

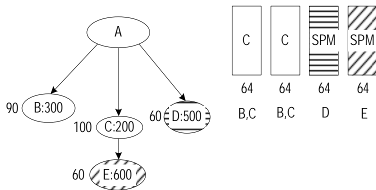

Figure 4: Example of loop allocation.

After a loop is examined, its parent loop will be added to $list_loops$ (line 14). The algorithm may at some later point examine it for more opportunities for reconfiguration. The algorithm therefore proceed one level at a time from leaves up to the root. If a loop is an internal node (line 5), then the algorithm will evaluate whether it is beneficial to allocate more SPM space from the free storage banks (line 9). The evaluation function we used is conservative and simple. If the reduction in cache size caused by the allocation of more space to SPM does not severely increase the instruction miss rate, then it is considered beneficial. The evaluation function takes the number of free storage banks for reconfiguration and the current loop as input. It returns the maximal number of additional SPM banks ($\#SPM_banks$) which can yield beneficial results.

Algorithm 1: Algorithm for determining dynamic reconfiguration and SPM instruction load points

```
   Input: Proc_list: Procedure list whose procedures have been intra-procedural
          optimized
   Output: Basic_block_list: list of instruction blocks of basic blocks assigned to
          SPM
   Variable list_loops : the list of loops;
   Variable list_child_loops : the list of loops;
1  Build Loop-Procedure Hierarchy Graph(LPHG)
2  Get_sizes_of_all_loops();
3  list_loops ⟵ all leaf loops;
4  foreach loop l in list_loops do
       if ((l is leaf loop) && (#iteration of l ≥ Thresh_hold)) then
           Annotate_reconfig_point_and_instrs_partitioned(l);
5      else if l is non-leaf loop then
6          list_child_loops ⟵ all child loops of l;
7          #banks_occcupied = # of banks configured as SPM for loops in
           list_child_loops;
8          #free_banks = #total_banks - #banks_occcupied;
9          #SPM_banks = evaluateConflict(#free_banks,
           child_loops_of_l ∪ l);
10         if (#SPM_banks !=0) then
11             Instr_alloc(list_child_loops ∪ l, SPM);
12             Update_reconfig_point(l);
           end
13     if (!list_loops.contain(parent_of(l))) then
14         list_loops.add_to_tail(parent_of(l)); //to traverse higher level loops later
       end
   end
15 Hoist_reconfig_position();
16 Insert_reconfig_and_code_loading_instructions();
17 return Proc_list;
```

Figure 4 is a example of how the algorithm evaluates conflicts and partitions the instructions. The left part of Figure 4 is a sample loop represented in LPHG, while the right are the four banks storage resource available. The algorithm first try to configure one bank as SPM and allocate it to loop E. Each of the left three child loops (i.e. B, C, D) can fit into the remaining three storage banks, i.e. there will not be any conflict. So, the algorithm will try to configure one more banks as SPM and move loop D, the loop with the next highest execution frequency, to it. Now, B and C, taken together, is smaller than the size of the two storage banks, and thus it is safe to take this configuration. If one more bank is configured as SPM, then there will only be one bank left to buffer the remaining loop and other code. It is therefore not beneficial to configure banks as SPM any more since severe cache conflicts will be caused with one of the loops.

The instruction allocation function allocates the frequently executed instructions inside the loop to the allocated SPM

(line 11). The instruction allocation function considers two factors. The first is the size of the loop. If it is larger than the size of the allocated SPM, then as many instructions as possible of the loop will be allocated to the SPM. The second consideration is the execution frequency of instructions. The most frequently executed instructions will be allocated to the SPM.

After instruction allocation, all reconfiguration points inserted in the child loops by the previous iteration will be deleted and a new reconfiguration point is added to the entry of the loop (line 12). This is because before a `SPM_load` instruction loads a block of code, the child loop should not load another instruction block. There can only be one block of instructions residing in the SPM. The instructions loaded to SPM are frequently executed. Therefore care must be taken

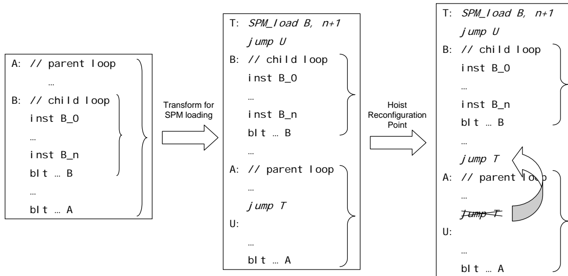

Figure 5: Code transformation for reconfiguration.

Once all the loops are traversed and the reconfiguration positions and instructions assigned to SPM have been decided, the instructions for reconfiguration and instruction loading are inserted. There is an important optimization that can be applied. The number of reconfigurations can be reduced by hoisting the reconfiguration point from inner loop to outer loop (line 15). If a loop does not have any sibling loops, the reconfiguration at its entry can be hoisted out to its parent loop. An example of this code transformation is shown in Figure 5. The `SPM_load` instruction loads a block of code into the SPM as well as set up the bound registers.

The last step in the algorithm (line 16), is to group all the instructions allocated to the SPM for each reconfiguration and insert the instructions used for reconfiguring the DRIM as well as loading the instruction blocks to SPM after each reconfiguration yielding the final transformed program.

5 Experimental Evaluation

5.1 Experimental Methodology

We used the Simplescalar/PISA 3.0d simulator [3] for our experiments. The full-featured simulator in the suite, `sim-outorder`, was modified to support DRIM. The cache line modeled in the simulator is 64 bytes, corresponding to 32 bytes in 4-byte instruction systems. The instruction memory hierarchy consists only of the L1 instruction buffer (i.e. DRIM) and the main memory. Our DRIM implementation has four banks of data storage, each of size 256 bytes. The latency of accessing DRIM is 1 cycle. The main memory is assumed to be pipelined. The latency of the first access to the main memory is 10 cycles, while that of the subsequent accesses is 2 cycles.

In our experiments, we used six application benchmarks from the MediaBench [12] and MiBench [8] suites. We compared the energy consumption and performance of executing each benchmark on two different architectures: (1) a baseline system comprising of a traditional 4-way associative instruction cache and (2) a DRIM based system.

We modeled the energy consumption of the memory hierarchy using the CACTI [17] model for $0.13\mu m$ technology. For the calculation of the energy consumption of DRIM, we included the logic elements that perform address checking and control the SPM. The energy consumption of loading a trace into the SPM is modeled as the number of SDRAM burst accesses up to the size of the trace. The dynamic energy consumption per access of different architectures is shown in Table 1. '1way', '2way', '3way' and '4way' represents the energy consumption of the cache portion when DRIM is configured as a combination of 1, 2, 3, or 4 banks cache and the SPM respectively. 'SPM' is the per access energy consumption for the SPM in DRIM. This is the sum of the energy consumption for one data bank of the 4-way associative cache and the energy overhead for accessing the SPM. The energy consumed by each burst access of SDRAM is 32.5 nJ [5].

	DRIM					
base cache	1way	2way	3way	4way	SPM	SDRAM
0.538	0.152	0.283	0.413	0.544	0.133	32.5

Table 1: Per access energy consumption (in nJ).

5.2 Performance Improvements and Energy Savings

Performance: The performance results are shown in Table 2. Compared to the baseline cache configuration, the decrease in the instruction cache miss rate provided by DRIM ranges from 0% to 40.7% for the benchmarks studied. The average improvement in the miss rate is 15.6%. This improvement comes from reconfiguring some storage banks to SPM and the mapping of the frequently executed instructions into the SPM for important loops. For the benchmarks `mpeg2-dec` and `mpeg2-enc`, there is no improvement on the miss rate because they are dominated by small size loops with very large number of iterations. Such benchmarks performs well on a pure cache architecture. As a result of the improvement in miss rates, the execution times of the applications are de-

creased by an average of 10.2%.

Benchmark	miss rate(%) base	miss rate(%) DRIM	miss rate(%) Imprv	execution cycles(K) base	execution cycles(K) DRIM	execution cycles(K) imprv(%)
gsm-dec	0.42	0.40	4.8	7,617	7,603	0.2
gsm-enc	6.10	3.62	40.7	70,076	47,633	32.0
g721-enc	3.09	2.43	21.4	381,509	331,266	13.2
susan-edge	2.76	2.03	26.4	2,346	1,962	16.4
mpeg2-dec	1.36	1.36	0.0	27,329	27,427	-0.4
mpeg2-enc	0.11	0.11	0.0	836,006	836,121	-0.0
average	-	-	15.6	-	-	10.2

Table 2: Miss rate and Performance

Energy consumption: The total energy consumption of the two instruction memory hierarchies are shown in Table 3. Compared to the baseline cache cofiguration, the reduction in the energy consumption provided by DRIM ranges from 14.3% to 65.2% for the benchmarks studied. The average reduction in the energy consumption is 41%.

	gsm-dec	gsm-enc	g721-enc	susan-edge	mpeg2-dec	mpeg2-enc
baseline(mJ)	8.39	98.34	558.52	3.27	37.39	1,019.7
DRIM(mJ)	4.60	53.84	336.3	2.08	32.04	354.75
improv(%)	45.2	45.2	39.8	36.5	14.3	65.2

Table 3: Energy consumption.

There are two major reasons for the reduction in energy consumption. First, the instruction cache miss rate has improved. The per access energy consumption of SDRAM is much higher than that of the cache and SPM. Thus, fewer cache misses will translate to energy savings. Second, the per access energy consumption of the SPM is lower than that of the cache. By configuring one or more instruction storage buffer as SPM and loading the frequently executed instructions into them during the program execution, significant energy savings can be obtained. For example, although there were no miss rate reduction for `mpeg2-dec` and `mpeg2-enc` (as shown in Table 2), there is actually energy savings. `mpeg2-enc` has a higher energy reduction than `mpeg2-dec` since its miss rate is very low and the energy consumption is dominated by on-chip instruction buffer accesses. By reconfiguring on-chip storage buffer banks as SPM, the total energy consumption is decreased significantly.

6 Conclusion

In this paper, we proposed a low power dynamically reconfigurable instruction memory hierarchy, called DRIM, for embedded systems. The on-chip instruction storage banks can be reconfigured as SPM or cache for different applications as well as different phases of the application's execution. We also developed a compilation flow to support DRIM. Our experimental results showed significant energy savings as well as satisfactory performance improvement. We believe that our approach is more flexible than previous schemes and can be easier applied to embedded systems.

References

[1] David H. Albonesi. Selective cache ways: on-demand cache resource allocation. In *Proceedings of MICRO-32*, pages 248–259, 1999.

[2] Rajeshwari Babakar, Stefan Steinke, Bo-Sik Lee, M. Balakrishnan, and Peter Marwedel. Scratchpad memory: A design alternative for cache on-chip memory in embedded systems. In *Proc. of CODES '02*, pages 73–78.

[3] Doug Burger and Todd M. Austin. The simplescalar tool set, version 2.0. *Technical Report #1342, University of Wisconsin-Madison Computer Sciences Department*, May 1997.

[4] Andhi Janapstya et. al. Hardware/software managed scratchpad memory for embedded system. In *ICCAD'04*, 2004.

[5] Aviral Shrivastava et. al. Compilation techniques for energy reduction in horizontally partitioned cache architectures. In *Proc. of CASES'05*, pages 90–96, 2005.

[6] Kondo M et. al. SCIMA: Software controlled integrated memory architecture for high performance computing. In *Proc. of ICCD'2000*, pages 105–111, 2000.

[7] M. Balakrishnan et. al. Reducing energy consumption by dynamic copying of instructions onto onchip memory. In *Proc. of ISSS'02*, pages 213–218, Kyoto, Japan, October 2002.

[8] Matthew R. Guthaus et. al. Mibench: A free, commercially representative embedded benchmark suite. *IEEE 4th Annual Workshop on Workload Characterization*, December 2001.

[9] Federico Angiolini et.al. A post-compiler approach to scratchpad mapping of code. In *Proc. of CASES '04*, pages 259–267, September 2004.

[10] Yanbing Li et.al. Hardware-software co-design of embedded reconfigurable architectures. In *Proc. of DAC '00*, pages 507–512.

[11] Zhiguo Ge, Weng Fai Wong, and Hock Beng Lim. A reconfigurable instruction memory hierarchy for embedded systems. In *Proc. of FPL'05*, pages 7–12, 2005.

[12] C. Lee, M. Potkonjak, and W. Mangione-Smith. Mediabench: A tool for evaluating multimedia and communications systems. *In Proceedings of the Micro-30*, December 1997.

[13] Karl Pettis and Robert C. Hansen. Profiling guided code positioning. In *Proc. of PLDI'90*, pages 16–27.

[14] Ranjiv A. Ravindran. Compiler managed dynamic instruction placement in a low-power code cache. In *Proc. of CGO'05*, pages 179–190.

[15] Tom van der Aa et al. Instruction buffering exploration for low energy vliws with instruction clusters. In *Proc. of ASP-DAC'04*, pages 824–829, 2004.

[16] Manish Verma, Lars Wehmeyer, and Peter Marwedel. Cache-aware scratchpad allocation algorithm. In *Proc. of DATE '04*, pages 1264–1269.

[17] Steven J. E. Wilton and Norman P. Jouppi. Cacti: An enhanced cache access and cycle time model. *IEEE Journal of Solid-State Circuits*, 31(5):677–688, May 1996.

[18] Chuanjun Zhang, Frank Vahid, and Walid Najjar. A highly configurable cache architecture for embedded systems. In *Proc. of ISCA-30*, pages 136–146, 2003.