A Novel Instruction Scratchpad Memory Optimization Method based on Concomitance Metric

Andhi Janapsatya†, Aleksandar Ignjatović†‡, Sri Parameswaran†‡
†School of Computer Science and Engineering, The University of New South Wales
Sydney, NSW 2052, Australia
‡NICTA, The University of New South Wales
Sydney, NSW 2052, Australia
{andhij,sridevan,ignjat}@cse.unsw.edu.au

ABSTRACT

Scratchpad memory has been introduced as a replacement for cache memory as it improves the performance of certain embedded systems. Additionally, it has also been demonstrated that scratchpad memory can significantly reduce the energy consumption of the memory hierarchy of embedded systems. This is significant, as the memory hierarchy consumes a substantial proportion of the total energy of an embedded system. This paper deals with optimization of the instruction memory scratchpad based on a novel methodology that uses a metric which we call the *concomitance*. This metric is used to find basic blocks which are executed frequently and in close proximity in time. Once such blocks are found, they are copied into the scratchpad memory at appropriate times; this is achieved using a special instruction inserted into the code at appropriate places. For a set of benchmarks taken from Mediabench, our scratchpad system consumed just 59% (avg) of the energy of the cache system, and 73% (avg) of the energy of the state of the art scratchpad system, while improving the overall performance. Compared to the state of the art method, the number of instructions copied into the scratchpad memory from the main memory is reduced by 88%.

1. Introduction

A designer looks beyond mere functionality of the required embedded system and optimizes for performance, energy consumption, and cost. Performance optimization allows for greater functionality, or the utilization of a lesser processor for the same task. Low energy consumption allows longer battery life, lower heat dissipation, and superior reliability. Reduced cost makes the system more competitive in the marketplace.

A customary method of improving performance and reducing energy consumption of a system is to use a cache. Despite the overall reduction in system energy, cache memory is known to consume up to half the total system energy. Instruction cache alone has been shown to consume up to 27% of total processor energy [1]. Thus, careful optimization in this area can reap rewards in terms of reduced energy consumption.

Prior research on optimization of instruction memory hierarchy for embedded systems saw the replacement of the instruction cache by a scratchpad memory (SPM) [2]. Even though cache and scratchpad are both made of SRAM cells, they operate differently. Cache is typically designed to improve performance of general-purpose processors, and is divided into tag RAM and data RAM. The tag RAM is compared against the instruction address requested, and if the instruction exists in the cache, it is sent to the processor, avoiding an access to the main memory. SPM, on the other hand, does not contain tag RAM, and forms a part of the main memory address map. By accessing the most frequently executed parts of the program from the SPM, one can reduce both the total execution time and the power consumption, by avoiding tag RAM storage and comparison. As embedded systems typically execute a restricted number of applications, there are opportunities for improving their performance that are not available in the case of general purpose systems. For example, patterns of execution of the blocks of the code can be understood by the use of profiling. Once these patterns are known, they can be used to fill the SPM with appropriate segments of code at appropriate moments [11]. This reduces both the total access time and the total energy consumption of the system. Additionally, energy consumption is reduced due to the fact that the processor requires fewer wait cycles.

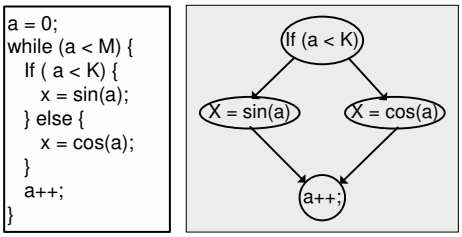
Selecting the correct code segments for placement in the SPM requires a careful analysis of the way such code segments are executed. All prior work in this area has focused upon loop analysis of the trace of a program as the method for finding the appropriate segments. Loop analysis has several drawbacks: (i) the structure and relationship of loops can be very complex; (ii) this structure can significantly vary for different inputs; (iii) the precise structure of loops is irrelevant for the placement of instructions in the SPM because only relative (temporal) proximity of executions matters, rather than the precise order of these instructions (as provided by the loop analysis) [12, 13, 14, 15].

In this paper we present, for the first time, an optimization method for utilizing instruction SPM that is based on an analysis of temporal correlation of instruction executions, rather than on loop analysis.

We introduce a class of metrics for estimating temporal proximity of consecutive executions of the same block of the code, and for estimating temporal proximity of interleaved executions of two different blocks of the code. The former is used to decide if a block should be executed from the SPM or from the main memory; the latter is used to decide if two blocks should be placed in SPM together or not. Such temporal information is gathered using a very efficient and adaptive algorithm whose parameters can easily be changed (various metrics for distance estimation). These metrics are used to estimate how correlated in time the execution of various blocks of code are, using an informative quantity that we call the *concomitance*. Our methods have a signal-processing flavor, because the trace is seen as a "signal" on which we perform a statistical, rather than structural analysis. Such analysis of the trace has proven to yield an algorithm for SPM placement with performance results that are not only superior to the previous state of the art, but that is also much simpler, more efficient, and adaptive to different types of applications. Recently we found out that a related, but somewhat cruder and less general idea has been used for cache management [16].

The benefits of using temporal proximity information is illustrated in the following example. Consider the "if-clause" shown in Figure 1(a) with the Control Flow Graph (CFG) shown in Figure 1(b). Looking at this code segment, for $K = 50$ and $M = 100$, a profiling

0-7803-9451-8/06/$20.00



(a) If-clause code segment (b) If-clause CFG

Figure 1: Motivational Example.

that takes into consideration temporal proximity of executions would find that the first 50 execution of the "if-clause" take the path containing $x = sin(a)$, and the next 50 execution of the "if-clause" take the path containing $x = cos(a)$. Thus, such algorithm will deduce that, since they are not interleaved in time, the block of instructions used for computing $x = sin(a)$ and the block of instructions used for computing $x = cos(a)$ can be placed onto SPM with overlap without degrading the performance. However, in a similar program, executions of $x = sin(a)$ and $x = cos(a)$ may be temporally interleaved. For example, if the "if-clause" is of the form $if(a =$ "*even number*"$)$, then the path with $x = sin(a)$ will be executed whenever a is an even number and the path with $x = cos(a)$ will be executed whenever a is an odd number. In such case the profiling will deduce that the executions of blocks used to compute these two functions alternate and consequently such temporally interleaved blocks must not be placed in the SPM with an overlap. Our concomitance metric is designed to distinguish between pairs of blocks whose executions are *likely* to be interleaved and blocks whose executions are *likely* to be separate in time, thus enabling proper placement onto the scratchpad. Clearly, if only frequency of execution was determined as in the state of the art [11], these two patterns of execution would be indistinguishable and CFG would simply inform the algorithm that each path was executed 50 times. This shows the importance of measuring appropriately the temporal proximity of executions of blocks of code and making such information available to the SPM placement algorithm.

The rest of this paper is structured as follows: section 2 summarizes the related work and our contribution; section 3 presents the SPM system architecture; section 4 defines the *concomitance* and presents techniques for building the *concomitance table*; section 5 presents the optimization method; Section 6 explains the experimental setup and presents the results; finally, section 7 states the conclusions reached in this paper.

2. Related Work and Contributions

Existing work on the utilization of SPM can be categorized into three areas: (i) use of SPM for data memory only; (ii) use for instruction memory only; or (iii) both. The SPM utilization can be further divided into two classes: statically managed use and dynamically managed use. With static management, the SPM is filled at load time and its content does not change during the execution; in dynamic management, the contents of the SPM is changed during the program run-time.

Existing works on cache optimization techniques rely on careful placement of instructions and/or data within the memory to ensure low cache miss rates. Cache optimization methods generally increase the program memory size [21, 22, 23, 24, 25, 26, 27, 28]. The use of SPM to replace cache memory has been shown to improve the performance and reduce energy consumption [2].

SPM optimization methods for data memory are presented in [3, 4, 5, 8, 6, 17, 18, 19, 20]. In optimizing for data memory, data access patterns were analyzed to find frequently accessed variables and constants. A dynamic management scheme of data SPM was first presented by Kandemir [5].

For static SPM, the most frequently executed basic blocks (e.g., loops) are kept within the SPM [2, 7]. Static SPM is simple, because it does not change its contents during execution. However, static SPM can be limiting, because it has to either keep all loops in the program by providing a large SPM to accommodate all the loops, or decide upon the most executed loops to be stored in a smaller SPM.

To overcome this limitation of static SPMs, dynamic SPMs update their contents during runtime. Steinke [9] used loop analysis to find loops within the program to dynamically allocate into SPM. They added a series of load and store instructions within the program to perform copying into SPM, resulting in greatly increased code size. In [11], Janapsatya et al. modified the processor architecture and inserted a custom instruction to perform copying into SPM. They used a graph partitioning procedure to perform loop analysis to find suitable loops within the program to place in the SPM. This reduced the number of instruction inserted, but was still dependent upon loop analysis. Their graph partitioning procedure was inefficient, since it used a global heuristic which performs badly for loops with basic blocks which are far apart in the program structure.

2.1 Our Contribution

In this paper, we define the notion of *concomitance* of blocks of code and use it to automatically identify blocks that are executed in clusters, each cluster consisting of many executions in close temporal succession. Such blocks are found to have high *concomitance* and will be placed in the SPM first. Further, pairs of basic blocks whose executions are interleaved are found to have high pairwise *concomitance* are thus simultaneously allocated to SPM. Hence, without the need to perform difficult loop analysis, we can identify which basic blocks are to be executed from SPM and which groups of instructions should be placed in the SPM simultaneously.

Our contribution includes replacement of difficult structural loop analysis of the program by an essentially statistical method for automated decision making regarding which basic blocks should be executed from SPM and, among them, which groups of basic blocks should be placed in the SPM simultaneously. Our method is very intuitive, flexible, and perspicuous, and is more akin to signal processing techniques rather than the existing algorithms for SPM management.

3. SPM System Architecture

3.1 Assumptions

The work presented in this paper was performed under the following assumptions:

- Size of scratchpad memory or cache memory is only available in powers of two and the size is known a priori. This is to enable better optimization.
- The applications and their execution patterns are known a priori.

3.2 Dynamic Instruction Scratchpad Memory

The architecture of the embedded system described in this work is presented in Figure 2. Our proposed SPM architecture uses a dynamic management scheme. The copying process can either use a series of load and store instructions as describe in [10] or use the special instruction method described in [11].

In our experimental setup, we decided to adopt the copying method described in [11] because it is the state of the art. To copy instructions into the SPM, the energy costs are incurred due to instructions fetch from DRAM, copying instructions into the SPM, and the energy cost of the memory copying hardware. Copying is performed by the custom copy instruction inserted within the program to inform the copy mechanism hardware when to start moving instruction into the SPM.

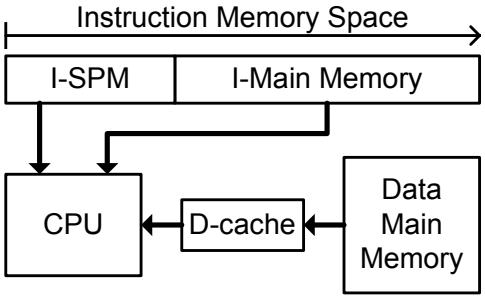


Figure 2: Embedded System Architecture.

Total energy cost of the system is given by summing the energy cost of all its components. For our dynamic scratchpad system, this is given by the following equation,

$$E_{system} = E_{CPU} + E_{SPM,insn} + E_{DRAM,insn} + E_{copy} + E_{cache,data} + E_{DRAM,data}$$

The parameters shown in the above equation are defined as follows:
E_{CPU} is the energy cost of the processor.
$E_{SPM,insn}$ is the sum of the energy costs of all instruction executed from scratchpad.
$E_{DRAM,insn}$ is the sum of the energy costs of all instruction executed from DRAM.
E_{copy} is the energy cost of all instructions copied into the SPM.
$E_{cache,data}$ is the energy cost of all data cache accesses.
$E_{DRAM,data}$ is the energy cost of all data cache misses.

The energy equation for calculating E_{copy} is given by,

$$E_{copy} = C \times E_{controller} + T \times (E_{SPM,write} + E_{DRAM,Burstread})$$

where C is the number of copy instructions executed, $E_{controller}$ is the energy cost of the copy mechanism, T is the total number of instructions copied into the SPM, $E_{SPM,write}$ is the cost of writing into the SPM, and $E_{DRAM,Burstread}$ is the cost for reading a sequential block of data from DRAM.

The heuristics behind our method can be explained as follows. If we are to place on to the SPM, a block of code a whose consecutive executions are often separated by execution of group g consisting of a large number of other distinct instructions, then either the part of the SPM occupied by a would not be utilized during the execution of g, or a would be overwritten by g and thus it would have to be reloaded. Consequently, we preferentially put in the SPM those blocks a that are executed frequently with a relatively small number of other distinct instructions executed between consecutive executions of a. Similarly, if two blocks of code a and b occur frequently in the trace in sequences of "sandwich forms" a,x,b,y,a or b,x,a,y,b such that x and y are some groups consisting in total of a relatively small number of distinct instructions of the program, then a and b should be allocated to the SPM in a non overlapping way. To formalize this heuristic, in section 4 we introduce the notion of *distance* between consecutive executions $e(b)$ and $e'(b)$ of a basic block b, as well as a notion of *concomitance* between basic blocks.

4. Concomitance

A basic block, by definition, is the largest chain of consecutive instructions that has the properties: (i), if the first instruction of the block is executed, then all instructions in the basic block will also be executed consecutively; and (ii), any instruction of the basic block is executed only as a part of the consecutive execution of the whole block.

The distance between two consecutive executions $e(b)$ and $e'(b)$ of a basic block b in the trace T of a run of a program is defined as follows. If between the executions $e(b)$ and $e'(b)$ of b there are no other occurrences of b in T, we count the number of ***distinct*** instruction steps executed between $e(b)$ and $e'(b)$, including b. We call this value *the distance between* $e(b)$ *and* $e'(b)$ and denote it by $d[e(b),\ e'(b)]$. For example, assume that "*bxyxyxyzxyxyb*" is a sequence of consecutive executions in a trace T, and that each of the basic blocks b, x, y and z contains ten distinct instructions; then the distance between $e(b),\ e'(b)$ is 40, because only x,y,z appear between the two executions of the basic block b (and we include b itself in the count).

The weight function is used to give a decreasing significance to the two consecutive executions of the same block that are further apart in the sense of the above notion of distance. Thus, it is a non-negative real function $W(z)$ that is decreasing, i.e. $u \leq v$ implies $W(u) \geq W(v)$.

The trace *concomitance* $\tau(a,b,T)$ gives information about how tightly interleaved the executions of two distinct basic blocks a and b in the trace T are. Thus, for a basic block a we consider all of its consecutive executions $e(a),\ e'(a)$ in the trace T, for which there exists at least one execution of the block b between the executions $e(a)$ and $e'(a)$; we denote such fact by $b \in [e(a),e'(a)]$. We now also reverse the roles of a and b, and define $\tau(a,b,T)$ by

$$\tau(a,b,T) = \sum_{\substack{b\in[e(a),e'(a)] \\ e(a)\in T}} W(d[e(a),e'(a)]) + \sum_{\substack{a\in[e(b),e'(b)] \\ e(b)\in T}} W(d[e(b),e'(b)])$$

Here $e(a) \in T$ in the sum means that $e(a)$ ranges over all executions of the basic block a that appear in the trace T. Note that for two distinct basic blocks $a,\ b$ the *concomitance* of these two blocks will be large just in case b is often executed between two consecutive executions of a that are a short distance apart, and/or if a is often executed between two consecutive executions of b that are also a short distance apart, in the sense of distance defined above.

The trace self-*concomitance* $\sigma(b,T)$ of a basic block b is a measure of how clustered consecutive executions of the block b are, and is defined as:

$$\sigma(b,T) = \sum_{e(b)\in T} W(d[e(b),e'(b)])$$

Thus, trace self-*concomitance* $\sigma(b,T)$ has a large value for those basic blocks b whose executions appear in clusters, with all successive pairs of executions within each cluster separated by short distances. Note that even if b is executed relatively frequently, but such executions of b are dispersed in the trace T rather than clustered, then self-*concomitance* $\sigma(b,T)$ will still be low. On the other hand, if for certain input a particular loop is frequently executed, then the trace self-*concomitance* $\sigma(a,T)$ of each basic block a from this loop will be large. Thus, the loop structure of a program is reflected in the *statistics* of the *concomitance* values if such statistics is taken over runs with sufficient number of inputs reasonably representing what is expected in practice. This is the motivation for the following definitions.

The *concomitance* $\overline{\tau}(a,b)$ of a pair of basic blocks a,b for a given probability distribution of inputs is the corresponding ***expected value*** of trace *concomitance* $\tau(a,b,T)$.

The self-*concomitance* $\overline{\sigma}(b)$ of a basic block b for a given a probability distribution of inputs is the corresponding ***expected value*** of trace self-*concomitance* $\sigma(b,T)$.

To conveniently use the *concomitance* and self-*concomitance* in our scratchpad placement algorithms, we construct the *concomitance* table by the following profiling procedure.

• *Chose a suitable weight function* $W(d)$. In our experiments so far we have studied two types of weight functions: $W(d) = \frac{M}{d}$ and $W(d) = e^{\frac{-d^2}{M}}$, where M is a constant depending on the size of the scratchpad.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
1	5																			
2	12	11																		
3	5	4.4	3.8																	
4	15	13	12	10																
5	5	4.4	3.8	3.4	3															
6	35	31	27	24	21	18														
7	1	0.9	0.8	0.7	0.6	0.5	0.5													
8	1	0.9	0.8	0.7	0.6	0.5	0.5	0.4												
9	8	7	6.1	5.4	4.7	4.1	3.6	3.2	2.8											
10	45	39	35	30	27	23	20	18	16	14										
11	13	11	10	8.8	7.7	6.7	5.9	5.2	4.5	4	3.5									
12	21	18	16	14	12	11	9.5	8.4	7.3	6.4	5.6	4.9								
13	1	0.9	0.8	0.7	0.6	0.5	0.5	0.4	0.3	0.3	0.3	0.2	0.2							
14	1	0.9	0.8	0.7	0.6	0.5	0.5	0.4	0.3	0.3	0.3	0.2	0.2	0.2						
15	1	0.9	0.8	0.7	0.6	0.5	0.5	0.4	0.3	0.3	0.3	0.2	0.2	0.2	0.2					
16	1	0.9	0.8	0.7	0.6	0.5	0.5	0.4	0.3	0.3	0.3	0.2	0.2	0.2	0.2	0.1				
17	1	0.9	0.8	0.7	0.6	0.5	0.5	0.4	0.3	0.3	0.3	0.2	0.2	0.2	0.2	0.1	0.1			
18	15	13	12	10	8.9	7.8	6.8	6	5.2	4.6	4	3.5	3.1	2.7	2.4	2.1	1.8	1.6		
19	5	4.4	3.8	3.4	3	2.6	2.3	2	1.7	1.5	1.3	1.2	1	0.9	0.8	0.7	0.6	0.5	0.5	
20	5	4.4	3.8	3.4	3	2.6	2.3	2	1.7	1.5	1.3	1.2	1	0.9	0.8	0.7	0.6	0.5	0.5	0.4

Figure 3: *Concomitance* Table.

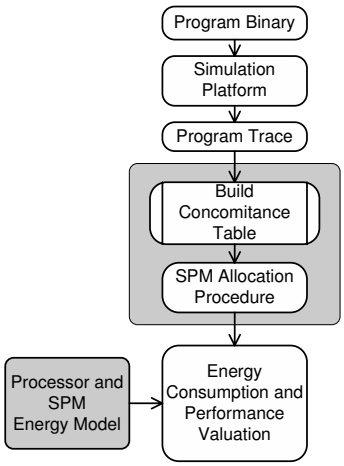


Figure 4: SPM Optimization Procedure.

• *Run the program* with inputs that reasonably represent the probability distribution of inputs expected in practice.

• *Calculate the average value* of *the trace self*-concomitance obtained from such runs, thus obtaining the self-*concomitance* value $\overline{\sigma}(b)$.

• *Set a threshold of significance* for the value of self-*concomitance* of basic blocks. The set S of all blocks with significant self-*concomitance* (i.e., larger than the threshold) is formed.

• *Calculate the* concomitance for all pairs of basic blocks from such set S, by finding the average of all trace *concomitances* obtained from the runs of the program and then form the corresponding table. Since the *concomitance* is commutative, such a table is symmetric and an example is shown in Figure 3; thus, we record only its lower left triangle. The self-*concomitance* $\overline{\sigma(b)}$ is conveniently placed on the diagonal of the table.

The size of the table is given by

$$Size = \frac{N*(N+1)}{2}$$

where N is the total number of basic blocks in the set S. Time complexity of the construction of the *concomitance* table is bounded by $2NT$, where T is the size of the trace.

5. SPM Optimization Procedure

The methodology for allocating basic blocks into the SPM is shown in Figure 4. A program binary is simulated along with its represented data to obtain the program trace. We then use our methodology to find segments in the program which are to be executed from the SPM. The methodology utilizes a SPM allocation procedure which uses the *concomitance* metric describe in section 4. This algorithm is shown in Figure 5.

The algorithm starts by building the *concomitance* table and the control flow graph (CFG) of the application. Each vertex in the CFG represents a basic block. The weight of each edge a represents the *concomitance* value $\overline{\tau}(a,b)$ of a and b, obtained from the *concomitance* table. Edges of this graph are sorted in increasing order with respect to *concomitance* value. We then select from the lowest *concomitance* value and evaluate whether two vertices connected by the edge belong to a sub-graph with size larger than SPM. We cut all edges which connect two vertices with total size larger than the SPM. The edge cut procedure is shown in Figure 6.

Build Control Flow Graph.
Build *Concomitance* table of elements in S.
Perform Edge Cut procedure based on *concomitance* value.
Evaluate the need for insertion of copy instruction on all cut edges.
Calculate energy cost of the resulting sub-graphs.
Calculate performance result of the sub-graphs.

Figure 5: SPM Allocation Procedure.

Construct one group containing all the vertices in S;
Sort the *concomitance* values of all pairs $\{a,b\}$ of basic blocks in ascending order;
Start from lowest *concomitance* value;
For each pair of basic blocks $\{a,b\}$ {
  if the resulting group size is larger than SPM size
    Cut the edge connecting a and b
}

Figure 6: Edge-cut Procedure.

For each subgraph, one or more copy instructions are to be added for copying the instructions within the subgraph into the SPM. Possible locations for adding copy instructions are the edges connecting subgraphs. To minimize the amount of copy instructions to be added, each edge is traversed up and a copy instruction is only added if another subgraph was possibly loaded into the SPM along the same execution path.

Time complexity of the SPM allocation procedure is bounded by N^3, where N is the total number of basic blocks in the set S.

6. Experimental Results

6.1 Setup

We simulated a number of benchmarks using the simplescalar/PISA 3.0d simulation environment [29] combined with DineroIV [30], to obtain memory access statistics. Power figures for the CPU were calculated using Wattch [31] ($0.18\mu m$). CACTI 3.2 [32] was used as the energy model for the cache memory. The energy model for the scratchpad memory was extracted from CACTI as in [2]. The DRAM power figures were taken from IBM embedded DRAM SA-27E [33]. We adopt the same Simplescalar CPU configuration and memory delay figures as described in [11] for ease of comparison of the results.

All benchmarks were obtained from the mediabench suite [34].The total number of instructions executed in each benchmark is tabulated in Table 1. The data cache memory is fixed at 4K bytes giving a large enough data cache to ensure all benchmark applications cause less than 1% data cache misses.

Table 2 shows the SPM access time, SPM access energy, cache ac-

App.	Prog. size	Total no. of insn. Exec.	Copy insn. inserted	Avg. no. of insn. copied into SPM	SMI added [11]	Avg. no. of Insn. copied into SPM[11]
rawcaudio	9182	6689768	4.6	1247	30.4	741972
rawdaudio	9384	12414463	4.6	1263	29.7	2704463
g721enc	11052	314594475	6.2	33751706	63	130722589
g721dec	11066	302967631	4.8	12844416	58	87833725
mpeg2enc	26808	1134231679	23.4	4385032	272.86	197787388

Table 1: Cost of adding and executing copy instructions.

Size (bytes)	Cache acc. time(ns)	SPM acc. time(ns)	ratio	Cache acc. energy(nJ)	SPM acc. energy(nJ)	ratio
512	1.19	0.74	1.61	1.37	0.18	7.61
1024	1.24	0.78	1.59	1.37	0.19	7.21
2048	1.30	0.83	1.57	1.39	0.20	6.95
4096	1.31	0.88	1.49	1.42	0.23	6.17
8192	1.34	1.05	1.28	1.49	0.29	5.14
16384	1.64	1.21	1.36	1.55	0.36	4.31

Table 2: Access time and energy consumption of static memory.

App.	SRAM size	Total cache misses assoc = 1	2	4	8	16	Total DRAM acc.[11]	Total DRAM acc.	% imp. cache	% imp. [11]
rawcaudio	1024	7408	6653	7818	8666	6807	3883	3004	59.4	22.6
	2048	7052	4629	2899	2846	2852	3427	2396	32.7	30.1
	4096	3931	4076	2334	2275	2240	3400	2092	24.1	38.5
	8192	2154	1981	1868	1847	1830	3400	2092	-8.5	38.5
	16384	2007	1841	1810	1799	1799	3400	1799	2.7	47.1
rawdaudio	1024	20899	26033	29336	30530	31148	31527	12488	53.7	60.4
	2048	14996	10389	4806	2914	2920	3470	2634	44.3	24.1
	4096	5465	5900	2398	2352	2322	3535	2160	29.8	38.9
	8192	2208	2035	1932	1915	1899	3535	2160	-8.5	38.9
	16384	2049	1898	1878	1867	1867	3535	1867	2.2	47.2
g721enc	1024	1.3E+8	1.3E+8	1.1E+8	1.1E+8	1.1E+8	1.5E+8	5.6E+7	52.6	62.8
	2048	1.1E+8	1.1E+8	1.1E+8	1.1E+8	1.1E+8	2.1E+8	5.6E+7	47.9	73.1
	4096	7.8E+7	8.3E+7	9.0E+7	1.0E+8	1.1E+8	3.3E+8	1.2E+8	-38.1	62.6
	8192	5.1E+7	2.1E+7	1.4E+7	1.1E+7	4.3E+6	7.5E+5	3.0E+5	97.2	59.5
	16384	3.9E+6	2.4E+6	1.1E+4	2.7E+3	2.7E+3	8.6E+3	2.7E+3	55.5	68.6
g721dec	1024	1.2E+8	1.2E+8	1.1E+8	1.1E+8	1.1E+8	1.3E+8	3.0E+7	73.5	77.1
	2048	1.0E+8	1.0E+8	1.0E+8	1.0E+8	1.0E+8	1.8E+8	3.9E+7	62.4	78.6
	4096	7.4E+7	8.0E+7	8.5E+7	9.2E+7	1.0E+8	1.1E+8	4.1E+7	51.8	62.5
	8192	4.1E+7	2.6E+7	8.6E+6	3.6E+6	3.7E+6	1.9E+5	4.7E+4	99.3	75.1
	16384	6.6E+4	3.5E+4	1.3E+4	3.6E+3	2.8E+3	6.8E+3	2.7E+3	58.8	60.0
mpeg2enc	1024	1.1E+8	1.4E+8	1.7E+8	1.8E+8	1.9E+8	2.4E+8	8.6E+7	43.6	64.2
	2048	2.7E+7	1.1E+7	1.1E+7	1.1E+7	1.1E+7	2.5E+7	5.2E+6	58.8	79.4
	4096	4.8E+6	4.1E+6	3.0E+6	2.8E+6	2.7E+6	6.7E+6	1.9E+6	43.4	71.9
	8192	1.9E+6	1.0E+6	7.6E+5	7.9E+5	8.4E+5	4.5E+6	7.7E+5	19.0	82.9
	16384	4.8E+5	3.7E+5	1.4E+5	1.1E+5	1.1E+5	2.8E+6	1.3E+5	21.1	95.4

Table 3: Total memory access Comparison.(Cache result shows the total cache misses. SPM total DRAM access is the total number of instructions executed from DRAM plus the number of instructions copied from DRAM to the SPM.)

cess time, and cache access energy [32] for an 8-way set associative cache (8-way is only shown here as an example). It can be seen that accessing SPM is approximately 1.5 times faster than a cache access and uses approximately 6 times less energy compared to an 8-way set associative cache. Instruction copying hardware cost is taken from [11] and stated to be at 2.94mW.

Experimental setup is shown in Figure 7. Comparison is made between results obtained in this work, work described in [11], and a conventional cache memory system.

6.2 Results

Table 3 showed a comparison of the total number of memory accesses. Performance and energy results measured from the experiments are shown in Table 4 and Table 5, and cost of adding copy locations within the program is shown in Table 1.

In Table 1, column 1 shows the application name, column 2 gives the size of the program, column 3 the number of instructions executed, column 4 gives the average number of copy locations to be inserted (average is taken from varying SPM sizes ranging from 1K bytes to 16K bytes), column 5 shows the average number of instructions that need to be copied into the SPM, column 6 presents the results from [11] showing the number of copy instruction (SMI) inserted into the program, and column 7 shows the total number of instruction copied into the SPM from [11]. Comparing figures in Table 1 column 5 and column 7 shows that our method significantly reduces the number of instructions to be copied into the SPM.

Table 3 compares the total number of memory accesses of a cache

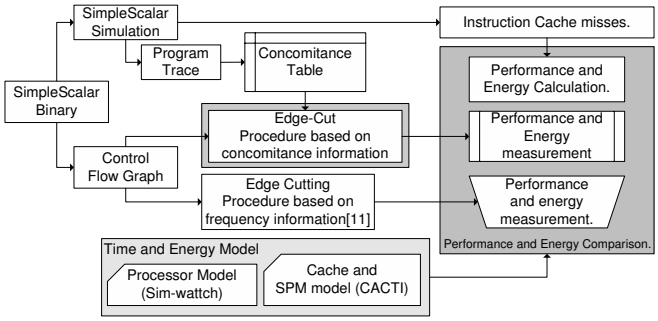


Figure 7: Experimental Setup.

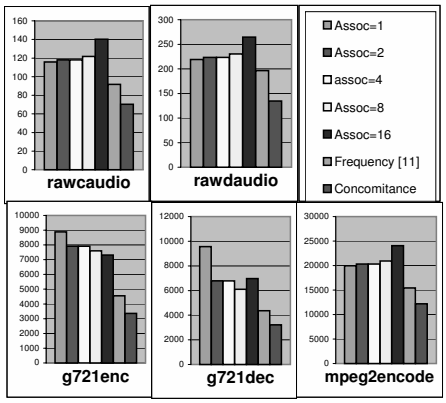


Figure 8: Energy Comparison for 8K bytes SRAM. (y-axis are energy in mJ)

system, the system presented in [11], and the system presented in this paper. Column 1 in Table 3 gives the application name; column 2 shows the cache or SPM size; column 3 to column 7 show the total number of cache misses for different cache associativities; column 8 gives the total DRAM accesses obtained from [11]; column 9 shows the total DRAM accesses obtained from our optimization method; column 10 shows the average percentage improvement of our method over the cache system; and column 11 the percentage improvement over the method described in [11]. In column 11, it is shown that our *concomitance* method reduces the total number of DRAM accesses by an average of 52.94% compared to the method described in [11]. When comparison is made with a cache based system (column 10), it is shown that the total number of DRAM accesses for our SPM system is not always less than the total number of cache misses. Despite the higher total DRAM accesses in a SPM system compared to the cache system, it does not always translate to worse energy consumption and worse performance compared to a cache system. This is because the energy and time cost per SPM access is far less compared to the energy and time cost per cache access (especially when compared to the energy and time cost of accessing a 16-way set associative cache.). It can also be noted that total number of DRAM accesses for a SPM system is comprised of both the number of instruction to be executed from DRAM and the total number of instruction copied from DRAM to SPM. Copying instructions from DRAM to SPM causes a sequential DRAM access which consumes less power and time compared to a random DRAM access that happens on each cache miss.

Table 4 shows energy comparison of our method with: (i) a cache system; and (ii) with the system presented in [11]. The table structure is identical to Table 3 except the comparison is now for energy. Figure 8 shows the energy improvement comparison between the cache system, our SPM allocation method, and the SPM allocation procedure described in [11] for all the benchmarks. The energy compari-

App.	SRAM size	Cache System's Energy (mJ) assoc = 1	2	4	8	16	SPM (mJ) [11]	SPM (mJ)	% imp. cache	% imp. [11]
rawcaudio	1024	101	105	105	113	130	69	59	45.9	13.6
	2048	103	106	106	117	132	73	64	43.2	12.6
	4096	109	111	111	118	138	77	66	43.2	14.3
	8192	116	118	118	122	140	92	70	42.5	23.3
	16384	140	139	139	147	165	99	75	48.5	24.2
rawdaudio	1024	194	201	201	217	249	157	116	45.1	26.1
	2048	197	200	200	222	249	161	122	42.4	24.2
	4096	207	211	211	223	260	170	127	42.3	25.0
	8192	219	223	223	230	264	196	135	41.7	31.4
	16384	265	263	263	277	310	210	143	47.7	31.6
g721enc	1024	25763	23507	23507	23753	24470	18397	10082	58.3	45.2
	2048	22782	22856	22856	23473	24123	22071	8920	61.6	59.6
	4096	18919	20190	20190	22273	24056	32151	14548	30.6	54.8
	8192	8880	7906	7906	7595	7314	4557	3356	57.5	26.4
	16384	7016	6568	6568	6932	7778	4832	3552	48.9	26.5
g721dec	1024	24830	22514	22514	22552	23315	16092	7185	68.9	55.4
	2048	21745	22026	22026	22589	23340	19071	6843	69.4	64.1
	4096	18141	19137	19137	20672	22841	13214	6968	64.9	47.3
	8192	9562	6779	6779	6104	6969	4357	3215	54.5	26.2
	16384	6382	6327	6327	6677	7493	4664	3422	48.3	26.6
mpeg2enc	1024	39931	44684	44684	49043	52774	34447	22183	51.6	35.6
	2048	19399	19849	19849	21853	24341	14118	11612	44.5	17.8
	4096	19292	19502	19502	20636	23970	13218	11636	43.1	12.0
	8192	19971	20294	20294	20941	24068	15421	12174	42.1	21.1
	16384	24032	23793	23793	25099	28151	16479	12907	48.1	21.7

Table 4: System's Energy Comparison.

App.	SRAM size	Cache System Execution Time (ms) assoc = 1	2	4	8	16	SPM (ms) [11]	SPM (ms)	% imp. cache	% imp. [11]
rawcaudio	1024	7.9	8.0	8.0	8.4	9.1	5.2	4.7	43.3	10.8
	2048	8.1	8.1	8.1	8.7	9.2	5.6	5.0	40.5	9.9
	4096	8.5	8.5	8.5	8.8	9.7	5.9	5.2	40.5	11.8
	8192	9.0	9.0	9.0	9.0	9.9	7.0	5.5	39.7	21.3
	16384	11.0	10.7	10.7	11.0	11.8	7.6	5.9	46.4	22.1
rawdaudio	1024	14.8	15.1	15.1	15.7	17.1	9.9	8.8	43.5	11.7
	2048	15.0	15.0	15.0	16.1	17.1	10.3	9.3	40.6	9.8
	4096	15.8	15.8	15.8	16.2	18.0	11.0	9.7	40.5	11.7
	8192	16.7	16.8	16.8	16.7	18.3	13.0	10.3	39.7	21.3
	16384	20.3	19.8	19.8	20.4	21.9	14.0	10.9	46.4	22.1
g721enc	1024	2016.8	1832.5	1832.5	1837.3	1864.5	1443.5	798.4	57.4	44.7
	2048	1782.8	1781.3	1781.3	1815.6	1837.2	1747.1	703.8	60.9	59.7
	4096	1479.5	1571.8	1571.8	1720.7	1831.8	2563.4	1147.7	29.4	55.2
	8192	690.9	607.3	607.3	567.7	517.3	335.0	261.8	55.8	21.9
	16384	545.1	502.2	502.2	516.9	554.1	355.4	277.1	47.0	22.0
g721dec	1024	1943.6	1754.9	1754.9	1744.0	1775.8	1260.1	568.2	68.3	54.9
	2048	1701.5	1716.5	1716.5	1747.1	1777.6	1507.9	538.6	68.9	64.3
	4096	1418.5	1489.5	1489.5	1596.0	1738.3	1034.2	548.0	64.4	47.0
	8192	744.5	519.2	519.2	451.5	492.2	319.1	250.6	52.7	21.5
	16384	495.5	483.6	483.6	497.8	533.6	342.2	266.8	46.4	22.0
mpeg2enc	1024	3142.5	3495.2	3495.2	3789.4	3978.2	2762.6	1742.9	51.0	36.9
	2048	1500.8	1509.6	1509.6	1616.0	1705.0	1123.1	899.1	42.5	19.9
	4096	1490.0	1479.5	1479.5	1516.2	1673.6	1048.2	900.3	40.9	14.1
	8192	1539.2	1538.7	1538.7	1534.4	1678.0	1221.2	942.9	39.7	22.8
	16384	1859.7	1812.6	1812.6	1865.2	1999.5	1300.3	1000.2	46.4	23.1

Table 5: System's Performance Comparison.

son shows that our method almost always performs better compared to the cache system, and superior results are seen when compared with results obtained from [11]. On average, the energy consumption by utilizing our method is 41.9% better than cache system energy, and 27.1% better than the method described in [11]. In particular our method is superior in cases where negative improvements over cache were shown in results from [11].

Performance result is shown in Table 5. Structure of the table is identical to Table 4; column 3 to column 7 shows the execution time of a cache based system; column 8 shows the performance results obtained from [11]; and column 9 shows our performance measurement results; column 10 shows the average performance improvement of our method over cache system; and column 11 shows performance improvement over method described in [11]. Our method improves the execution time by 40.0% compared to cache and 23.6% compared to the method described in [11].

Thus it is clear from the results that the method described here is a feasible method for dynamic SPM allocation. We show that the number of copy instructions inserted are far fewer than the state of the art. In addition, for the applications shown here, we show performance improvement and energy savings.

7. Conclusions

In this paper we have proposed a method to reduce energy and improve performance of an embedded system, containing an instruction scratchpad. Our system relies on a new metric called *concomitance*, which is used to identify basic blocks that should be placed together in the scratchpad. This method results in embedded systems with lower energy and higher performance, compared to a standard cache system and state of the art scratchpad instruction partitioning algorithm.

8. References

[1] J. Montanaro et al., "A 160MHz, 32b, 0.5W CMOS RISC microprocessor," *JSSC,* vol.31(11), pp. 1703-1712, 1996.

[2] R. Banakar et.al., "Scratchpad Memory: A Design Alternative for Cache On-chip Memory in Embedded Systems," *CODES,* 2002.

[3] O. Avissar and R Barua, "An Optimal Memory Allocation Scheme for Scratch-Pad-Based Embedded Systems," *ACM Trans. on Embedded Computing Systems,* vol. 1, pp. 6-26, 2002.

[4] P.R. Panda, "Efficient Utilization of Scratch-Pad Memory in Embedded Processor Applications," *European Design and Test Conference, Proceedings of,* 1997.

[5] M. Kandemir et.al., "Dynamic Management of Scratch-Pad Memory Space," *DAC,* 2001.

[6] M. Kandemir and A. Choudhary, "Compiler-Directed Scratch Pad Memory Hierarchy Design and Management," *DAC,* 2002.

[7] F. Angiolini et.al., "Polynomial-Time Algorithm for On-Chip Scratchpad Memory Partitioning," *CASES,* 2003.

[8] S. Udayakumaran and R. Barua, "Compiler-Decided Dynamic Memory Allocation for Scratch-Pad Based Embedded Systems," *CASES,* 2003.

[9] S. Steinke et.al., "Assigning Program and Data Objects to Scratchpad for Energy Reduction," *DATE,* 2002.

[10] S. Steinke et.al., "Reducing Energy Consumption by Dynamic Copying of Instructions onto Onchip Memory," *ISSS,* 2002.

[11] A. Janapsatya et.al., "Hardware/Software Managed Scratchpad Memory for Embedded System," *ICCAD,* 2004.

[12] G. Ramalingam, "On Loops, Dominators, and Dominance Frontier," *PLDI,* 2000.

[13] P. Havlak, "Nesting of Reducible and Irreducible Loops," *ACM Transactions on Programming Languages and Systems,* 1997.

[14] V. C. Sreedhar et. al., "Identifying Loops using DJ Graphs," *ACM Transactions on Programming Languages and Systems,* 1996.

[15] B. Steensgaard, "Sequentializing Program Dependence Graphs for Irreducible Programs," *Technical Report MSR_TR-93-14,* Microsoft Research, Redmond, Washington, October 1993.

[16] N. Gloy and M. D. Smith, "Procedure Placement Using Temporal-Ordering Information," *Programming Languages and Systems. ACM Transactions on,* Vol. 32, No. 5, Pages 977-1027, September 1999.

[17] P. Grun et. al., "Access Pattern-Based Memory and Connectivity Architecture Exploration," *Embedded Computing Systems. ACM Transactions on,* Vol. 2, No. 1, Pages 33.73, February 2003.

[18] A. Ramachandran and M. F. Jacome, "Xtream-Fit: An Energy-Delay Efficient Data Memory Subsystem for Embedded Media Processing," *DAC,* 2003.

[19] M. Verma et. al., "Dynamic Overlay of Scratchpad Memory for Energy Minimization," *CODES+ISSS,* 2004.

[20] M. Kandemir et. al., "Exploiting Scratch-Pad Memory Using Presburger Formulas," *ISSS,* 2001.

[21] S. Parameswaran and J. Henkel, "I-CoPES: fast instruction code placement for embedded systems to improve performance and energy efficiency," *ICCAD,* 2001.

[22] P. P. Chang and et.al., "IMPACT: an architectural framework for multiple-instruction-issue processors," *Computer Architecture News,* vol. 19, no. 3, 1991.

[23] S. McFarling, "Program optimization for instruction caches," *ASPLOS,* 1989.

[24] S.McFarling, "Procedure merging with instruction caches," *SIGPLAN Notices,* vol. 26, no. 6, 1991.

[25] P. Panda and et.al., "Memory Organization for Improved Data Cache Performance in Embedded Processors, *ISSS,*, 1996.

[26] P. Panda and et.al., "A Data Alignment Technique for Improving Cache Performance," *ICCD,*, 1997.

[27] H. Tomiyama and H. Yasuura, "Optimal code Placement of Embedded Software for Instruction Cache," *EDAC*, 1996.

[28] S. Bartolini and C. A. Prete, "A cache-aware program transformation technique suitable for embedded systems," *Information and Software Technology 44(13),* 2002.

[29] D. Burger and T. M. Austin, "The SimpleScalar Tool Set, Version 2.0," *TR-CS-1342,* University of Wisconsin-madison, June 1997.

[30] J. Edler and M. D. Hill, "Dinero IV Trace-Driven Uniprocessor Cache Simulator," *http://www.cs.wisc.edu/ markhill/DineroIV/.*

[31] D. Brooks et.al., "Wattch: A Framework for Architectural-Level Power Analysis and Optimizations," *ISCA,* 2000.

[32] P. Shivakumar and N. P. Jouppi, "Cacti 3.0: An Integrated Cache Timing, Power, and Area Model," *Technical Report 2001/2,* Compaq Computer Corporation, August, 2001. 2001.

[33] IBM Microelectronics Division, "Embedded DRAM SA-27E," *http://ibm.com/chips*, 2002.

[34] C. Lee et.al., "MediaBench: A Tool for Evaluating Multimedia and Communications Systems," *IEEE MICRO 30,* 1997.