Hardware/Software Managed Scratchpad Memory for Embedded System

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

Abstract **In this paper, we propose a methodology for energy reduction and performance improvement. The target system comprises of an instruction scratchpad memory instead of an instruction cache. Highly utilized code segments are copied into the scratchpad memory, and are executed from the scratchpad. The copying of code segments from main memory to the scratchpad is performed during runtime. A custom hardware controller is used to manage the copying process. The hardware controller is activated by strategically placed custom instructions within the executing program. These custom instructions inform the hardware controller when to copy during program execution. Novel heuristic algorithms are implemented to determine locations within the program to insert these custom instructions, as well as to choose the best sets of code segments to be copied to the scratchpad memory. For a set of realistic benchmarks, experimental results indicate the method uses 50.7% lower energy (on average) and improves performance by 53.2% (on average) when compared to a traditional cache system which is identical in size. Cache systems compared had sizes ranging from 256 to 16K bytes and associativities ranging from 1 to 32.**

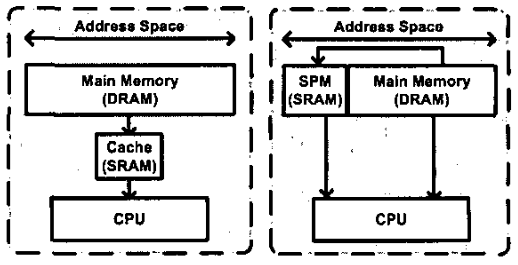
1. Introduction

Designers of embedded systems constantly strive to improve performance and reduce energy consumption. Low energy systems extend battery life, reduce cooling costs, and decrease weight. Improved performance allows cheaper components to be utilized while still meeting all necessary deadlines. Shutting down parts of the processor [1], voltage scaling [2], specialized instructions [3], feature size reduction [4], and additional cache levels [5], are some of the techniques used to reduce energy in embedded systems.

One area which consumes substantial amounts of energy in a typical processing machine is the instruction memory system. For example, the StrongArm SA110 consumes 27% of its total power in instruction cache alone [6]. Despite a complex tag-comparison system, cache is widely used to improve performance, since cache (made of SRAM) has faster access time compared to DRAM. For general purpose systems, usage of cache reduces total execution time and lowers energy consumption.

Embedded systems differ from general-purpose systems by executing the same application or class of applications repeatedly. Knowledge of an embedded application's profile can be well understood through extensive simulations. To reduce power in embedded systems, profile knowledge applied to a system with scratchpad memory (called SPM henceforth) instead of instruction cache, has been shown to be useful [7].

A SPM, made of SRAM cells, is a memory array consisting of only decoding and column circuitry logic [7]. The typical instruction memory hierarchy consists of a main memory (constructed with DRAM cells) and a SPM as shown in Figure 1(b). The content of the SPM is updated through the bus connecting the DRAM to the SPM. Traditional instruction memory hierarchy with cache (SRAM) and main memory (DRAM) is shown for comparison in Figure 1(a). In the SPM configuration, it is possible for the CPU to access the DRAM directly, unlike in cache architecture where the CPU has access to DRAM only through the cache.



(a) Cache Configuration (b) SPM Configuration

Figure 1: Instruction Memory Hierarchy and Address Space Partitioning

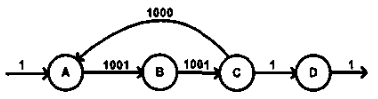


Figure 2: Motivational Example

To use the SPM configuration effectively, the compiled code is initially loaded into DRAM. Whenever, a code segment which is repeatedly utilized is encountered (a special instruction inserted by the compiler alerts the system of this encounter), that segment is copied to the SPM, and the segment proceeds to be executed from the SPM. Other segments which are run just a few times, or are run sporadically, can be executed directly from the DRAM, reducing copying costs.

The motivation for utilizing SPM instead of cache is shown in the following example. We model an application fragment as a directed graph (shown in Figure 2) where vertices are basic blocks and edges are transitions from one basic block to another. A basic block is a group of instructions, which are unconditionally and consecutively executed. The weights on the edges give the number of transitions from one block to another, found a priori through profiling (static analysis).

For the graph given in Figure 2, it is known that execution of basic blocks B & C will always follow basic block A. Hence, B & C can always be loaded when loading A into SPM, eliminating the need for tag comparisons. Instead of a tag-comparison, a special instruction to load basic block A, B, and C into SPM can be inserted just before the execution of basic block A. If the SPM was small and could only contain blocks A & B, then C could be directly executed from DRAM. Cache on the other hand would have thrashed (between block C and whatever is presently in cache), resulting in slower execution time and increased energy consumption.

In this paper we present a novel architecture, containing a special hardware unit to manage dynamic copying of code segments from main memory to SPM. We describe the architecture, and show the use of a specially created instruction which triggers the copying from main memory to SPM. We further set forth heuristic algorithms which rapidly evaluate which sets of code segments should be copied to SPM, and where to place the specially created instructions for maximum impact. The whole system was evaluated using benchmarks from mediabench and UTDSP.

The rest of this paper is organized as follows: section 2 describes previous works on SPM and presents the contributions of our work; section 3 introduces our strategy for using the scratchpad memory; section 4 formally describes the problem; section 5 presents the algorithm for partitioning the application code; section 6 provides the experimental setup and the results; and section 7 concludes this document.

2. Related Work and Contributions

Various schemes for managing SPM have been introduced, and can be broadly divided into two schemes: static and dynamic. Both these schemes are usually applied to the data section of a program, code section of the program or both.

Static management refers to the partitioning of data or code prior to execution, and storing the appropriate partitions in the SPM with no transfers occurring between these memories during the execution phase (occasionally the SPM is filled from the main memory at start up). Memory words are fetched directly from either of the memory partitions to the processor. Dynamic management on the other hand moves highly utilised memory words from the slow memory to the SPM, before transferring to the processor, thereby allowing code or data with a larger memory footprint than the SPM to utilise the SPM. Effective partitioning of memory footprint for use in static SPM, and dynamic SPM have been performed in the recent past.

In 1997, Panda et.al. [9] presented a scheme to statically manage a SPM for use as data memory. They describe a partitioning problem for deciding whether data variables should be located in SPM or DRAM (accessed via the data cache). Their algorithm maximizes cache hit rate by allowing data variables to be executed from SPM. Avissar et.al. [8] presented a different static scheme for utilizing SPM as data memory. They presented an algorithm that can partition the data variables as well as the data stack among different memory units, such as SPM , DRAM, and ROM to maximize performance.

In 2001, Kandemir et.al. introduced dynamic management of SPM for data memory [10]. Their memory architecture consisted of SPM and DRAM accessed through cache. To transfer data from DRAM into SPM, they created two new software data transfer calls; one to read instructions from DRAM and another to write instructions into SPM.

In 2003, Udayakumaran and Barua [12] improved upon [10] and presented a more general scheme for dynamic management of SPM as data memory. Udayakumaran's method analyzes the whole program at once and aims to exploit locality throughout the whole program instead of individual loop nest.

In 2002, Steinke et.al. [13] presented a case to statically manage SPM for both instruction and data memories. In 2003, Angiolini et.al. presented another SPM static management scheme for data memory and instruction memory. They presented a polynomial time algorithm for partitioning data and instructions into DRAM and SPM [11].

In [14], Steinke et al. presented a dynamic management scheme of instruction memory utilizing a SPM and main memory. For each instruction to be copied into the SPM, they inserted a load and a store instruction. The load instruction brought the instruction to be copied

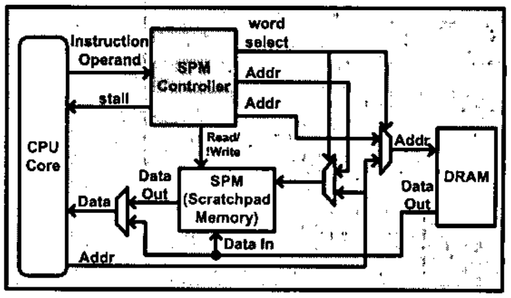


Figure 3: SPM system architecture

from main memory to processor, and store instruction stored it back into SPM. They created an algorithm to determine which segments should be copied to SPM in order to obtain maximal energy savings. The algorithm starts by analyzing the program to identify highly executed program parts. It determines possible locations in the program to insert load and store instructions. Finally, the best sections to be loaded are ascertained and the load store instructions are actually inserted. The algorithm is based upon an Integer Linear Programming (ILP) and solved using an ILP solver. They conducted experiments with small applications (bubble sort, heap sort, 'biquad' from DSP stone benchmark suit, etc.).

The work described in this paper dynamically copies code segments (instructions only) of a program to SPM from main memory, and allows execution from both the SPM and the main memory. The work radically departs from the work in [14], by using a special hardware controller to manage the copying of code from DRAM to the SPM. This special hardware controller is activated by a single instruction for each code segment to be copied, whereas the model proposed in [14], needed two extra instructions for every instruction to be copied. In [14], the authors used an ILP formulation to search for segments to be copied to the SPM. In contrast, we created a fast heuristic algorithm to find the best segments of the program to be copied from DRAM to SPM allowing the system to handle large realistic embedded applications. Note that our system requires certain hardware modification, while the system in [14] does not.

Our work improves upon the state of the art in the following ways:

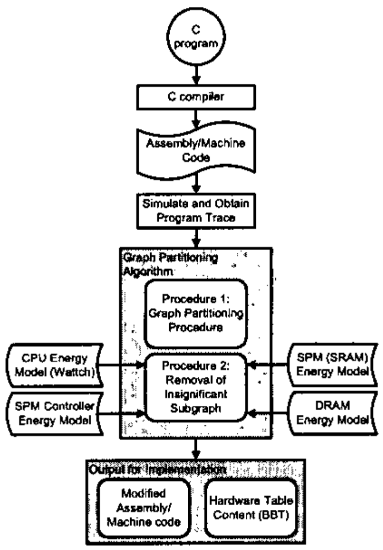
- we present a novel architecture to perform dynamic management of instruction code between SPM and main memory.
- for the first time the problem of partitioning instruction codes between SPM and DRAM is modelled as a graph problem.
- a novel graph partitioning algorithm to separate the instruction code into segments(all segments are initially in main memory and those segments which are highly utilised are marked, so that they can be copied into SPM and executed) to reduce overall energy dissipation.

We evaluated the system by using realistic embedded applications and show performance and energy comparisons with processors of various cache sizes and differing associativities.

3. Our Approach

3.1 Strategy

We propose a methodology to utilize SPM through hardware and software modification. To manage the SPM, a SPM controller and a new instruction called SMI (Scratchpad Managing Instruction) is implemented. Figure 3 shows the block diagram of the SPM controller, SPM, DRAM, and the CPU. The SPM controller is responsible for stalling the CPU when copying from DRAM to SPM, and to provide memory addresses of both DRAM and SPM during the copying process. The SMI is used to activate the SPM controller. SMIs are inserted within the program and is executed whenever the CPU encounters a SMI.



C program
C compiler
Assembly/Machine Code
Simulate and Obtain Program Trace
Graph Partitioning Algorithm
Procedure 1: Graph Partitioning Procedure
Procedure 2: Removal of Insignificant Subgraph
CPU Energy Model (Wattch)
SPM Controller Energy Model
SPM (SRAM) Energy Model
DRAM Energy Model
Output for Implementation
Modified Assembly/ Machine code
Hardware Table Content (BBT)

Figure 4: Methodology for Implementing the SPM

Figure 4 shows the steps for implementing the proposed methodology. The methodology operates on the assembly/machine code level; hence a C program (or any other language program) is first compiled into assembly/machine code. Profiling and tracing (using modified simplescalar/Pisa 3.0d [15]) is performed on the assembly/machine code to obtain an accurate picture of the program behavior. This behavior is depicted as a graph.

An algorithm (see Section 5) utilizing the graph, and energy models of the various hardware blocks is implemented. The algorithm is used to determine appropriate locations to insert SMIs within the program, and to decide which basic blocks should be executed from SPM, and which should be in DRAM.

In implementing the algorithm for insertion of the SMIs, the following assumptions are taken.

- Size of the SPM is known during compilation of program. This is a reasonable assumption since the program is to execute in an embedded application where the underlying hardware is known a priori. This assumption is made to allow for greater optimization. However, if a number of processors with differing sizes of SPM have to be serviced by the same binary source, it is possible to ship several binaries, and the suitable one can be applied to the particular embedded system.
- Program size is larger than the SPM size. This is a valid assumption, for if it is not, then the whole program can be allocated to SPM.
- Size of largest basic block is less than or equal to the SPM size. This assumption is quite valid in embedded systems where basic blocks are usually small enough to fit into small SPMs. However, if the basic block is too large for the SPM it can be broken up into smaller granules such that each granule will fit into the SPM.
- Each instruction is always either executed from SPM or DRAM. This ensures that it will never be necessary either to have duplicates of codes or to alter branch destinations during the execution of a program.
- Program trace is an accurate depiction of program execution. This assumption is reasonable when a sufficiently large input space has been applied. The amount of profiling needed to obtain a particular confidence interval is given in [16].
- Higher level caches are not available for use. Once again, in an embedded system, where frequently there is no cache at all, it is unlikely that more than a single level of cache (in this approach the L1 cache is replace by a SPM) is available for use. However, having higher level caches does not reduce the effectiveness of the approach.

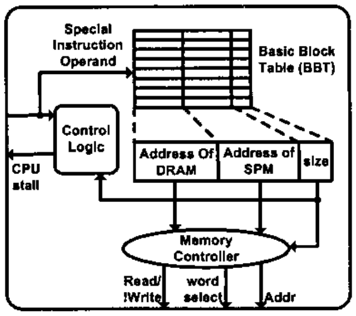


Figure 5: Architecture of the SPM controller

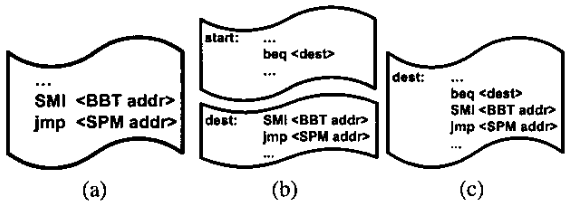


Figure 6: Condition for insertion of SMI into programs.

The strategy for using the SPM is as follows: at the start of a program, the first instruction is always fetched from DRAM. When a SMI is fetched, the CPU will activate the SPM controller. The SPM controller will then stall the CPU and starts copying instructions from DRAM into SPM. After copying is complete, the SPM controller will release the CPU, and the CPU will continue to fetch and execute instructions from DRAM. Whenever a branching instruction leads program execution to the SPM, it will then start to fetch instructions from the SPM.

3.2 Hardware Implementation

In this approach the CPU is modified by the addition of a SPM-controller, addition of a SMI, and the SPM. The system does not have a level-1 instruction cache. The micro-architecture of the SPM-controller is shown in Figure 5 and includes the Basic Block Table (BBT). The BBT stores the following static information: start addresses of the basic blocks to be read from within the DRAM; how many instructions are to be copied; and the start addresses for storing the basic block into SPM. Content of the BBT is filled by the algorithm shown in Figure 4.

3.3 Software Modification

The new instruction, SMI, is an instruction with a single operand. The operand of the SMI represents the address of a BBT entry. For a 32 bit instruction with 16 bit operand, it is possible to have up to 65536 entries in the BBT allowing 65536 unique SMIs to be inserted into a program.

There are three conditions under which the code could be modified. These are: before an unconditional branch; destination of a conditional branch; and just after a conditional branch. The modifications are elaborated in the following paragraphs.

Insert a SMI just before an unconditional branch (Figure 6(a)). Instructions to be executed by the execution of the unconditional branch can be copied into SPM using the SMI. The destination address of the unconditional branch is to be altered so that it points to the correct address within the SPM.

Add a SMI as destination of a conditional branch (Figure 6(b)). In the case of a conditional branch (which tests to be true), a SMI is

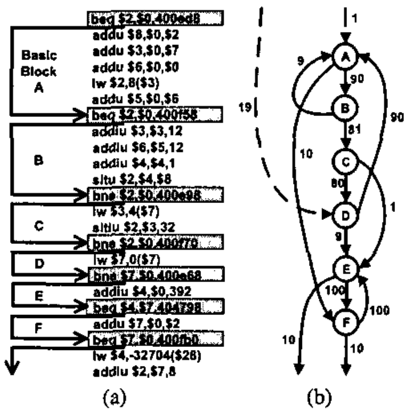

```
beq $2,$0,400ed8
addu $8,$0,$2
addu $3,$0,$7
addu $6,$0,$0
lw $2,8($3)
addu $5,$0,$6
beq $2,$0,400f58
addiu $3,$3,12
addiu $6,$5,12
addiu $4,$4,1
sltu $2,$4,$8
bne $2,$0,400e98
lw $3,4($7)
sltiu $2,$3,32
bne $2,$0,400f70
lw $7,0($7)
bne $7,$0,400e68
addiu $4,$0,392
beq $4,$7,404798
addu $7,$0,$2
beq $7,$0,400fb0
lw $4,-32704($28)
addiu $2,$7,8
```

Figure 7: An example of identified basic block and its graph representation.

added to copy the basic block to be executed. An extra jump instruction is added following the SMI to transfer program execution to the SPM.

A SMI instruction is added just after a conditional branch instruction (Figure 6(c)). In the case of a conditional branch (which tests to be false), a SMI is inserted to copy the basic block to be executed following the branch instruction. An extra jump instruction needs to be added to transfer program execution to the SPM.

An extra branch instruction may also need to be added at the end of the basic block if execution flow is required to jump to another location within memory. If there is an unconditional branch at the end of the basic block, then the branch destination can be simply altered, or modified as in 1. If there are conditional branches then we may have to modify as per 2 or 3 depending upon whether they are to be executed from DRAM or SPM.

The algorithm shown in Figure 4, outputs the modified software including the addition of any extra branching instructions.

4. Problem Description

The problem can be formally described as follows. Given any program, it can be represented as a graph. The vertex represents a basic block. The edges represent the program execution path. Weight of each vertex represents the number of instructions within a basic block and the frequency of each edge represents the number of times this execution path is taken.

For the code fragments given in Figure 7(a), the basic blocks are identified as shown. The graph representation of the code fragment is shown in Figure 7(b).

The problem is to find vertices within the graph (i.e. the basic blocks of instructions) where one should insert the custom instruction, SMI. This instruction ensures that one or several subsequent vertices are copied from DRAM to SPM. These insertions are done in places that optimize energy consumption.

To locate strategic points within a program to insert SMIs, we transform our problem into partitioning a graph into subgraphs where each edge connecting subgraphs would be an optimal location to insert a SMI. It is easy to see that the "Minimum Cut into Bounded Sets Problem" [17] is P-time Turing reducible to our graph partitioning problem; thus our problem has no polynomial time algorithm, and we must use a heuristic approximation algorithm.

5. Algorithm Description

We implemented an algorithm consisting of two procedures based on various heuristics: Graph Partitioning Procedure and Removal of Insignificant Subgraphs Procedure.

5.1 Graph Partitioning Procedure

```
Sort list of edges in ascending frequency.
Repeat until total weight of each subgraph is less than N. {
    Eliminate set of edges that have frequency equal to the first edge in
    the ordered list.
    For each subgraph that is a connected component
    calculate the total vertex weight; {
        If the total weight of this subgraph is larger than N,{
            Form a list of its edges preserving the ordering
            from the initial list.
        }
    }
}
```

Figure 8: Graph partitioning procedure

Consider the following problem.

Assume that we are given:

1. *a graph with a set of vertices V and a set of edges E,*
2. *for each vertex $v \in V$ we are given a vertex weight $W(v)$ and for each edge $e \in E$ an edge frequency $F(e)$,*
3. *a constant N. (size of SPM)*

Find a partition of the graph, such that each subgraph has a total vertex weight less than or equal to N, and the total frequency of the edges connecting the subgraph is minimal.

A heuristic approximation algorithm for the above partitioning problem is implemented as shown on Figure 8.

Start by ordering the edges in ascending frequencies, i.e. such that the edge at the top of the list is one of the edges with the minimal frequency. We eliminate all the edges of the graph which have such a minimal frequency. This usually induces a partition of the graph into several subgraphs; the total weight of vertices for each of these subgraphs is calculated and compared with N. We then apply the same procedure to all subgraphs whose total vertex weight is larger than N, and the procedure stops when all of the subgraphs have a total vertex weight smaller or equal to N. Complexity of the graph partitioning procedure is $\Theta(e * log(e))$, where e is the number of edges in the graph.

If we identify the eliminated edges connecting the subgraphs as the locations to insert the special instructions for copying from DRAM to SPM, this would result in a quite sub-optimal solution, because there would be subgraphs that are too small to be worthwhile to copy and execute from SPM. For example, the graph partitioning procedure can produce results such as the one shown in Figure 10. The dotted lines indicate the partitions of the graph. It can be seen that partition 2 resulted in basic block B as a subgraph and it will only ever be executed once. Thus, it is not cost effective to first read it from DRAM and copy it to SPM, since one can just read it and execute it directly from DRAM. To search and remove these type of subgraphs, another heuristic procedure is implemented.

5.2 Removal of Insignificant Subgraphs Procedure

Such a heuristic procedure is shown in Figure 9. It starts by calculating the energy cost of executing each subgraph S from DRAM and the energy cost of executing such subgraph from SPM. Energy cost of executing S from DRAM $E_{DRAM}(S)$ is calculated using

$$E_{DRAM}(S) = DRAM_{acc} * \sum_{v \in S} \left(W(v) * \sum_{e \in IE(v)} F(e) \right) \quad (1)$$

where $DRAM_{acc}$ is the energy cost of a single DRAM access per instruction, and $IE(v)$ is the set of all edges incoming into the vertex v.

Equation 2 shows the energy cost $E_{SPM}(S)$ of executing instructions within the subgraph S from SPM, including the cost of accessing the DRAM once and copying to SPM.

```
For each subgraph S {
    Calculate energy cost $E_{DRAM}(S)$ of executing S from DRAM,
    Calculate energy cost $E_{SPM}(S)$ of executing S from SPM,
    if $E_{SPM}(S) < E_{DRAM}(S)$
        classify S to be executed from SPM,
    else
        classify S to be executed from DRAM.
}
For each edge e eliminated in the graph partitioning algorithm
    if e is an incoming edge for a SPM subgraph S {
        Do depth-first search to determine if every path
        whose all intermediary subgraphs are DRAM subgraphs,
        and which terminates with e must have its beginning
        at the same SPM subgraph S;
            if no, then insert SMI to the edge e.
    }
}
```

Figure 9: Removal of insignificant subgraph procedure

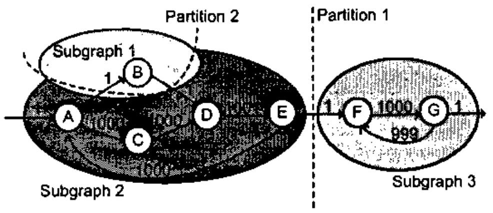


Figure 10: Subgraph Representation

$$E_{SPM}(S) = SPM_{acc} * \sum_{v \in S} \left(W(v) * \sum_{e \in IE(v)} F(e) \right) + E_{special}(S) \quad (2)$$

SPM_{acc} is the energy cost of a single SPM access per instruction. $E_{special}(S)$ is defined as

$$E_{special}(S) = \sum_{e \in IE(S)} F(e) * \left(DRAM_{acc} * \sum_{v \in S} W(v) \right) + q * E_{branch} \quad (3)$$

$E_{special}(S)$ is energy cost of executing the special instruction including reading from DRAM and copying to SPM and energy cost E_{branch} of adding a branch instruction if necessary. This happens if the proper execution of the branching instruction should change the flow from DRAM to SPM or vice versa. If such an extra branching instruction is necessary, constant q is equal to 1, else q is 0.

We denote by $IE(S)$ the set of all edges that are incoming into S from other subgraphs; clearly such edges are among those which were eliminated in the previous procedure.

The energy calculation is used to classify which subgraph is to be executed from SPM and which subgraph to be executed from DRAM; only subgraphs S with $E_{SPM}(S) < E_{DRAM}(S)$ will be executed from SPM. We call such a subgraph a SPM-subgraph.

The rest of the procedure inserts the SMI as follows. For every SPM-subgraph, it will examine all incoming edges to this subgraph that were eliminated in the graph partition procedure (subsection 5.1). We determine if such an edge is included in a path either from another SPM-subgraph, or from the start of the program, and only in such cases a SMI is inserted. This means that if all paths including this edge emanate from the same SPM-subgraph, SMI need not be inserted. The complexity of this procedure is $\Theta(V^3)$, where V is the number of vertices in the graph.

For example, in the graph shown in Figure 10, there are three subgraphs obtained from partition 1 and partition 2. Thus, partition 2 produced a subgraph consisting of the basic block B only. Assume that the energy cost estimation using equation 1 and 2 classified basic block B to be executed from DRAM. Consider the edge from B to D. All paths containing the edge from B to D also contain the edge from A to B. Since A belongs to the same SPM subgraph as D, by our procedure no SMI will be inserted in the edge from B to D.

By removing subgraphs that are not worthwhile to be executed from SPM, and minimizing the number of edges requiring a special instruction, we are able to minimize the number of special instructions to be added, and reduce any unnecessary replacement of instructions already in the SPM.

Parameters	Configuration
Issue Queue	16 entries
Load/Store Queue	16 entries
Fetch Queue	4 entries
Fetch/Decode Width	4 inst. per cycle
Issue/Commit Width	4 inst. per cycle
Function Units	4 IALU, 1 IMULT, 2 FPALU, 1 FPMULT
L1 ICache	8way, 2 cycles
L1 DCache	32KB, 2way, 1 cycle
Memory	100 cycles for first chunk, 10 cycles the rest

Table 1: SimpleScalar configuration.

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.

6. Experimental Results

6.1 Setup

We simulated a number of benchmarks using simplescalar/PISA 3.0d simulation environment [15], to obtain memory access statistics. Power figures for the CPU were calculated using Wattch [18] ($0.18\mu m$). CACTI 3.2 [19] was used as the energy model for the cache memory. The energy model for the scratchpad memory was extracted from CACTI as in [7]. The DRAM power figures were taken from IBM embedded DRAM SA-27E [20]. The configuration for the simulated CPU is as shown in Table 1.

Table 2 shows the SPM access time, SPM access energy, cache access time, and cache access energy [19] 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 caches.

The benchmarks were obtained from the mediabench suite [21]. Total number of instructions executed in each benchmarks is tabulated in Table 3.

6.2 Hardware Cost

In cache, tag RAMcells keep track of the entries in the cache while in the SPM system proposed here, the tags are replaced by the BBT. Each BBT entry needs to store one DRAM address, one SPM address, and the number of instructions to be copied. Since the most significant bits of the SPM address is known from the memory map (as shown in Figure 1(b)), only the least significant bits of the SPM address need to be stored within the BBT.

For example, given DRAM size of 2M bytes, SPM size of 1K bytes, and instruction size of 8 bytes. There exists enough space to store up

Application	total instructions executed	average number of SMI added	average number of SMI exec.	% increase in number of insn. exec.	ave. number of insn. copied from DRAM to SPM	ave. number of insn exec. from SPM	average insn. copy rate (%)	average cache miss rate (%)
adpcmenc	6689222	30.4	63392	0.95	741972	6470247	11.1	13.9
adpcmdec	12413917	29.7	84527	0.68	2704463	11565318	21.8	23.4
G721enc	314532454	63.0	2514381	0.80	130722589	285502170	41.6	31.0
G721dec	302896836	58.0	1875956	0.62	87833725	273549063	29.0	30.6
pegwitkey	13483894	81.6	20230	0.15	506801	12889791	3.8	11.8
pegwitenc	18941701	120.0	41143	0.12	1131996	26388136	6.0	26.7
pegwitdec	33700290	113.1	20261	0.11	601407	14642705	1.8	27.2

Table 3: Cost of adding and executing SMIs.

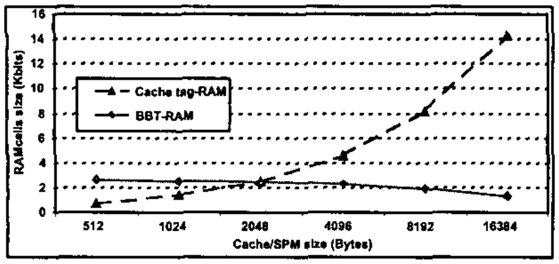


Figure 11: Cache tag-RAM size compared to BBT size in bits.

to 256K instructions within the DRAM and 128 instructions within the SPM. Thus, the number of instructions that need to be copied ranges from 1-128 requiring 7 bits per entry, Each DRAM address requires 18 bits per entry (to manage 256K instructions), and each SPM address requires 7 bits per entry. In total each BBT entry requires enough space to store 32 bits.

Figure 11 shows comparison between the number of bits in a cache tag RAMcells and the average size of the BBT for different size cache/SPM with memory size of 2M bytes. The average size of the BBT is calculated from all the benchmarks shown in Table 3. From Figure 11, it can be seen that cache tag RAMcells grows exponentially as the cache size grows, but the BBT size decreases as the SPM grows. BBT size decreases as SPM size increases because with a larger SPM, fewer SMIs are needed.

A SPM controller with BBT size of 128 entries (4096 Bits) was implemented in Verilog. The SPM controller is a finite state machine that accesses the BBT and forwards the output of the BBT as DRAM and SPM memory addresses. Power estimation of the SPM controller was done using a commercial power estimation tool with the following settings: $ClockFreq. = 500MHz$; $Voltage1.8V$; using a $0.18\mu m$ technology. Power estimation showed that it consumed 2.94mW of power. This is in comparison with cache tag power of 159mW, obtained from CACTI [19], for a $0.18\mu m$ 64 bytes of direct map cache. The power would progressively increase for greater ways of associativity. In addition, we also synthesised the SPM controller using, Synplify Pro [22], for a Xilinx Virtex 800K gate FPGA. The result shows that the SPM controller would occupy less than 1% of the FPGA resource.

6.3 Special instructions execution cost

With traditional cache architecture, DRAM is accessed every time a cache miss is encountered. In our SPM architecture, DRAM is accessed whenever a SMI is executed and whenever a code segment is executed from DRAM.Table 3, provides the application name in column 1, number of instructions executed for the application in column 2, average number of SMIs added in column 3, the average number of SMIs executed in column 4, the percentage of executed instructions which were SMIs in column 5, the average number of instructions which were copied from DRAM to SPM in column 6, average number of instructions which were executed from SPM in column 7, the average instruction copy rate in column 8 (which is (column 6 / Column 2) * 100), and for comparison purposes, the average cache miss rate is shown in column 9. Note that the average figures above refers to all SPMs sized from 256 to 16K bytes, and cache figures are for 256 to 16K bytes with associativities ranging from 1 to 32. From results shown in Table 3, it is seen that execution of SMI causes at most 0.95% increase in the number of instruction to be executed. On average 33 instructions are copied to the SPM every time a SMI is executed. Once an instruction is copied, on average it executes 4.3 times from the SPM.

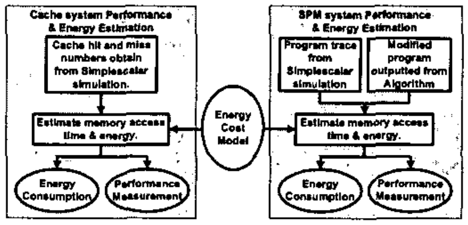

Cache system Performance & Energy Estimation
Cache hit and miss numbers obtain from Simplescaler simulation.
Estimate memory access time & energy.
Energy Consumption
Performance Measurement
Energy Cost Model
SPM system Performance & Energy Estimation
Program trace from Simplescaler simulation
Modified program outputted from Algorithm
Estimate memory access time & energy.
Energy Consumption
Performance Measurement

Figure 12: Experimental Steps for Evaluating Performance and Energy Improvement.

6.4 Performance and Energy Saving

Performance of the memory architecture is evaluated by calculating the total memory accesses for a complete program execution. Estimation of memory access time is possible due to known SPM access time, cache access time, DRAM access time, hit rates of cache and number of times the SPM contents are changed. Figure 12 shows the performance and energy estimation methodology.

The left shaded box in Figure 12 shows the steps to estimate the cache access time, $cache_{access\_time}$ and the cache energy consumption, $cache_{energy}$. $cache_{access\_time}$ is calculated using equation 4

$$\begin{aligned} cache_{access\_time}(ns) =& (cache_{hit} + cache_{miss}) * cache_{time}(ns) + \\ & cache_{miss} * DRAM_{time}(ns) \end{aligned} \tag{4}$$

where $cache_{hit}$ is the total number of cache hits, $cache_{miss}$ is the number of cache misses, $cache_{time}$ is the access time to fetch one instruction from the cache, and $DRAM_{time}$ is the amount of time spent to access one DRAM instruction.

In the right shaded box in Figure 12, the performance and energy estimation for the SPM architecture is shown. We estimate the total memory access time for the SPM architecture using equation 5,

$$\begin{aligned} SPM_{access\_time}(ns) =& SPM_{exe} * SPM_{time}(ns) + \\ & SMI_{exe} * copy_{size} * DRAM_{time}(ns) + \\ & DRAM_{exe} * DRAM_{time}(ns) \end{aligned} \tag{5}$$

where SPM_{exe} is the number of instructions executed from SPM, SPM_{time} is the amount of time needed to fetch one instruction from the SPM, SMI_{exe} is the total number of times all SMIs are executed, $copy_{size}$ is the total number of instructions copied from DRAM to SPM during program runtime, and $DRAM_{exe}$ is the number of instruction executed from DRAM.

App Name	SRAM Size (bytes)	Memory access time comparisons							Energy consumption comparisons						
		SPM (sec.)	Cache Associativity(sec.)					ave. % savings	SPM (Joule)	Cache Associativity(Joule)					ave. % savings
			1	2	4	8	16			1	2	4	8	16	
g721enc	1K	1.280	2.017	1.833	1.833	1.837	1.864	31.793	16.377	25.763	23.507	23.507	23.753	24.470	32.325
	2K	1.562	1.783	1.781	1.781	1.816	1.837	13.201	19.783	22.782	22.856	22.856	23.473	24.123	14.793
	4K	2.367	1.479	1.572	1.572	1.721	1.832	-44.779	29.727	18.919	20.190	20.190	22.273	24.056	-40.719
	8K	0.096	0.691	0.607	0.607	0.568	0.517	84.005	1.600	8.880	7.906	7.906	7.595	7.314	79.804
	16K	0.113	0.545	0.502	0.502	0.517	0.554	78.464	1.834	7.016	6.568	6.568	6.932	7.778	73.692
g721dec	1K	1.099	1.944	1.755	1.755	1.744	1.776	38.748	14.104	24.830	22.514	22.514	22.552	23.315	39.061
	2K	1.325	1.701	1.716	1.716	1.747	1.778	23.495	16.809	21.745	22.026	22.026	22.589	23.340	24.774
	4K	0.842	1.418	1.490	1.490	1.596	1.738	45.571	10.834	18.141	19.137	19.137	20.672	22.841	45.791
	8K	0.089	0.744	0.519	0.519	0.452	0.492	83.746	1.508	9.562	6.779	6.779	6.104	6.969	79.162
	16K	0.109	0.495	0.484	0.484	0.498	0.534	78.211	1.778	6.382	6.327	6.327	6.677	7.493	73.235
pegwitenc	1K	0.122	0.169	0.167	0.167	0.170	0.170	27.484	1.600	2.161	2.141	2.141	2.200	2.242	26.525
	2K	0.115	0.153	0.152	0.152	0.155	0.157	24.921	1.515	1.964	1.949	1.949	2.014	2.082	23.916
	4K	0.116	0.151	0.151	0.151	0.152	0.158	23.908	1.525	1.929	1.947	1.947	1.977	2.088	22.907
	8K	0.050	0.090	0.085	0.085	0.084	0.087	42.402	0.682	1.157	1.105	1.105	1.116	1.192	39.926
	16K	0.014	0.056	0.054	0.054	0.056	0.060	74.233	0.233	0.717	0.711	0.711	0.752	0.838	68.723

Table 4: Table of results showing percentage memory access time improvement and percentage energy improvement.

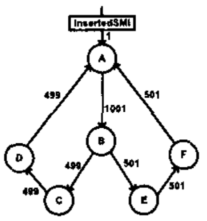


Figure 15: Example of a condition where SPM capacity miss can occur.

Application Name	SRAM size (Bytes)	Memory access time savings (%)	Energy savings (%)
bubble sort	256	31.2	30.9
heap sort	256	26.8	26.7
quick sort	256	26.4	26.3
biquad	256	35.5	31.6

Table 5: % memory access time savings and % energy savings of a 256 bytes SPM compared to 256 bytes 4-way associative cache.

SPM memory access time is compared to cache memory access time using the following equation.

$$\%_{improvement} = 100-((SPM_{access\_time}(ns)/cache_{access\_time}(ns))*100) \quad (6)$$

Table 4 shows the comparison between the SPM architecture and cache architecture memory access time and energy consumption (results for the other benchmarks are not shown due to space constraint). The bar graphs seen in Figure 13 show the percentage difference of SPM memory access time compared to cache memory access time. For example, given the memory access time for *g721enc* application with 1K bytes SPM compared to a 16-way associative 1K bytes cache from Table 4; the bar graph in Figure 13(a) shows that the SPM memory access time is 31.8% faster compared to the cache memory access time. This was calculated using equation 6 ($\%_{improvement} = 100-(1.280/1.864)*100$). Calculation of the average $\%_{improvement}$ for all the benchmarks shows that SPM can achieve performance improvement of 51.6% on average over cache architecture.

Although shorter instruction memory access time does not always imply a shorter execution time due to data memory access time and the time required by the CPU to execute multi-cycle instructions. For the purposes of evaluation, we minimized the effect of data memory access on the execution time by setting a large data cache so that data cache miss rates are less than 0.01%. Thus, the data cache has a very small to negligible miss rate, hence minimal effect on the program execution time.

For multi-cycle instructions, it is not possible to accurately estimate the execution time without the extra knowledge of which multi-cycle instructions were executed. We perform comparisons between cache execution time obtain from Simplescalar simulation to cache memory access time obtain from equation 4. We found that on average 9.5% error is seen between the two values with the maximum error of 17%.

Some of the results show performance degradation for SPM compared to cache (e.g. *G721enc* result for a 4K bytes SPM size compared to a 4K bytes cache). Investigation of the result found that it is due to capacity miss condition that is unique to the SPM case and does not occur when cache is used. Such a capacity miss condition is shown in the following example (Figure 15). Assume vertices represent basic blocks, edges represent the execution paths, each vertex has an equal number of instructions, and the SPM can only contain 4 vertices concurrently. By using SPM with a SMI inserted as shown in the example, we have to choose 4 vertices to be loaded into SPM and the remaining 2 vertices will execute from DRAM. If a cache had been used, depending upon the path taken, the cache will contain the correct 4 vertices currently executed and will cause very few cache misses. We classify this condition as a SPM capacity miss.

Energy consumption comparison is shown in Table 4 and Figure 14. The cache energy consumption is calculated using equation 7,

$$\begin{aligned} cache_{energy} =& (cache_{hit} + cache_{miss}) * E_{cache} + \\ & cache_{miss} * E_{DRAM} + cache_{access\_time} * E_{CPU} \end{aligned} \quad (7)$$

where E_{DRAM} is the energy cost per DRAM access and E_{CPU} is the energy consumed by the CPU.

The SPM energy is calculated using equation 8,

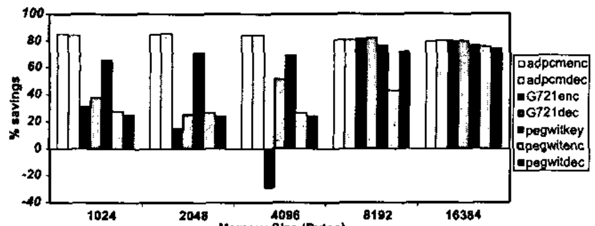
$$\begin{aligned} SPM_{energy} =& E_{SPM} * SPM_{exe} + E_{DRAM} * DRAM_{exe} + \\ & K * E_{SMI} + J * E_{branch} + SPM_{access\_time} * E_{CPU} \end{aligned} \quad (8)$$

where E_{SPM} is the energy cost per SPM access, E_{SMI} is the energy cost per execution of the special instruction, K is the total number of times all special instruction is executed, E_{branch} is the energy cost for any additional branch instructions, and J is the total number of times any additional branch instruction is executed.

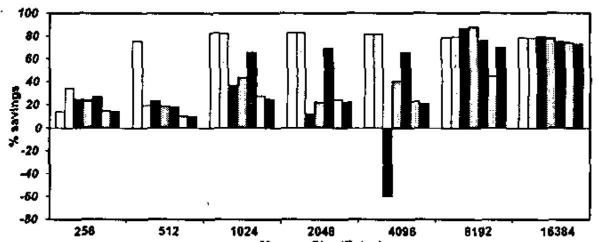
Percentage difference between the SPM energy consumption and the cache energy consumption is calculated using the following equation.

$$\%_{energy\_improvement} = 100-((SPM_{energy}/cache_{energy})*100) \quad (9)$$

Results of the percentage energy improvement is shown in Figure 14. It was calculated that the SPM architecture can achieve on average 49.6% energy reduction compared to energy consumption of a cache architecture.

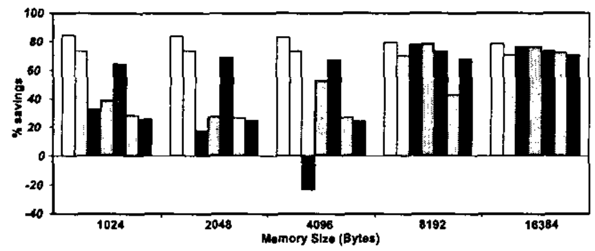


(a) SPM vs 16-way Cache

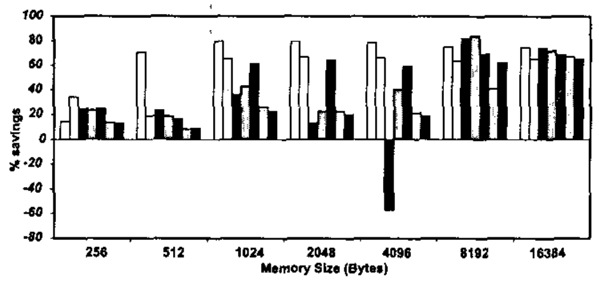


(b) SPM vs dm Cache

Figure 13: % savings of SPM memory access time over cache memory access time. (Legend shown in Figure 13(a) indicates orderings of bar-graphs from left to right.)



(a) SPM Energy vs 16-way Cache energy



(b) SPM Energy vs dm Cache energy

Figure 14: % savings of SPM execution energy over cache execution energy. (Legend for this figure is as shown in Figure 13(a).)

In addition to the benchmarks shown in Table 4, Figure 13, and Figure 14, we also implemented various *sort* algorithms taken from [23] and the *biquad_N_sections* from DSPstone benchmark suite. Results of these benchmarks are shown in Table 5. These benchmarks are similar to the ones used in [14]. From [14], the result for *heap sort* for the same SRAM size shows 4.6% performance improvement and 3.5% energy savings; and result from *quick sort* [14] with 256 bytes SRAM shows 10% performance improvement and 16% energy savings. Despite values in Table 5 showing our methodology achieves higher energy savings and performance improvement compared to methods in [14], the two results from [14] and Table 5 should not be compared directly due to the different underlying architectures used. Some of this energy savings are probably due to the differing data set size, while the rest is due to the reduced SMIs inserted.

7. Conclusions

We have presented a method to lower energy consumption and improve performance of embedded systems. The presented methodology uses SPM to store highly utilized code segments. By using a custom hardware SPM controller to dynamically manage the SPM, we have successfully avoided the need to insert many instructions into a program for managing the content of the SPM. Instead, we implemented heuristic algorithms to strategically insert custom instructions, SMI, for activating the hardware SPM controller. Experimental results show that our SPM scheme can lower energy consumption by an average of 50.7% compared to traditional cache architecture, and performance is improved by an average of 53.2%.

8. References

[1] K. Lahiri et al., "Communication Architecture Based Power Management for Battery Efficient System Design," *DAC*, 2002.

[2] J. Luo and N. K. Jha, "Power-profile Driven Variable Voltage Scaling for Heterogeneous Distributed Real-time Embedded Systems," *VLSI Design*, 2003.

[3] F. Sun et al., "Custom-Instruction Synthesis for Extensible-Processor Platforms," *TCAD*, 2004.

[4] J. M. Rabaey, "Digital Integrated Circuits: A Design Perspective," *Prentice Hall*, 1996.

[5] J. Kin et al., "The Filter Cache: An Energy Efficient Memory Structure," *IEEE Micro*, 1997.

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

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

[8] 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.

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

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

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

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

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

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

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

[16] P. Bartlett et.al., "Profiling in the ASP Codesign Environment," *Journal of Systems Architecture*, vol. 46, no. 14, pp. 1263-1274, Elsevier, Netherlands, Dec. 2000.

[17] M. R. Garey and D. S. Johnson, "Computers and Intractability," *W. H. Freeman and Company*, New York, 2000.

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

[19] 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.

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

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

[22] Synplicity Inc, "Synplify Pro," *http://www.synplicity.com*, 2004.

[23] R. Sedgewick, "Algorithm In C," *Addison-Wesley*, 3rd Edition, 1998.