Exploiting Statistical Information for Implementation of Instruction Scratchpad Memory in Embedded System

Andhi Janapsatya, *Member, IEEE,* Aleksandar Ignjatovič, and Sri Parameswaran *Member, IEEE*

***Abstract*— A method to both reduce energy and improve performance in a processor-based embedded system is described in this paper. Comprising of a scratchpad memory instead of an instruction cache, the target system dynamically (at runtime) copies into the scratchpad code segments that are determined to be beneficial (in terms of energy efficiency and/or speed) to execute from the scratchpad. We develop a heuristic algorithm to select such code segments based on a metric, called *concomitance*. *Concomitance* is derived from the temporal relationships of instructions. A hardware controller is designed and implemented for managing the scratchpad memory. Strategically placed custom instructions in the program inform the hardware controller when to copy instructions from the main memory to the scratchpad. A novel heuristic algorithm is implemented for determining locations within the program where to insert these custom instructions. For a set of realistic benchmarks, experimental results indicate the method uses 41.9% lower energy (on average) and improves performance by 40.0% (on average) when compared to a traditional cache system which is identical in size.**

***Index Terms*— Scratchpad Memory, Embedded system.**

I. Introduction

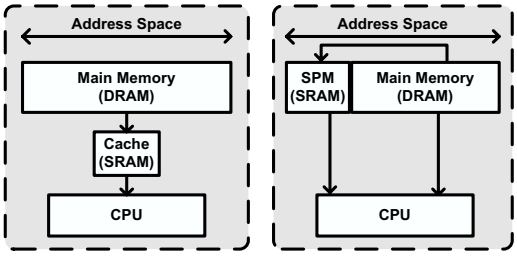
Processors are increasingly replacing gates as the basic design block in digital circuits. This rise is neither extraordinary nor rapid. Just as transistors slowly gave way to gates, and gates to more complex circuits, processors are progressively becoming the predominant component in embedded systems. As microprocessors become cheap, and the time to market critical, it is but a natural progression for designers to abandon gates in favor of processors as the main building components. The utilization of processors in embedded systems gives rise to a plethora of opportunities to optimize designs, which are neither needed nor available to the designer of an Application Specific Integrated Circuit (ASIC).

One critical area of optimization is the reduction of power in systems while increasing or at least maintaining the performance. The criticality stems from usage of embedded systems in battery powered devices as well as the reduced reliability of systems which operate while emanating excessive amounts of heat. Examples of existing techniques for achieving reduced energy consumption in embedded systems are: shutting down parts of the processor [1], voltage scaling [2], addition of application specific instructions [3], [4], feature size reduction [5], and additional cache levels [6].

Cache memory is one of the highest energy consuming components of a modern processor. The instruction cache memory alone is reported to consume up to 27% of the processor energy [7]. We present a method for reducing instruction memory energy consumed in an embedded processor by replacing the existing instruction cache (Figure 1(a)) with instruction scratchpad memory (SPM, Figure 1(b)). SPM consumes less energy per access compared to cache memory because SPM dispenses with the tag-checking which is necessary in the cache architecture. In embedded systems design, the processor architecture and the application that is to be executed are both known a priori. It is therefore possible to extensively profile the application and find code segments which are executed most frequently. This information can be analyzed and decisions made about which code segments should be executed from the SPM.

The authors are with the School of Computer Science & Engineering, University of New South Wales, Sydney, NSW 2052, Australia (e-mail:andhij, ignjat, sridevan@cse.unsw.edu.au)

Aleksandar Ignjatovic and Sri Parameswaran are also with NICTA, The University of New South Wales



(a) Cache Configuration (b) SPM Configuration

Fig. 1. Instruction Memory Hierarchy and Address Space Partitioning

Various schemes for managing SPM have been introduced in the literature, and can be broadly divided into two types: static and dynamic. Both these schemes are usually applied to the data section and/or the code section of the program. 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 to the processor directly from either of the memory partitions. Dynamic management on the other hand moves highly utilized memory words from the main memory to the SPM, before transferring to the processor, thereby allowing the code or data executed from the SPM to have a larger total memory footprint than the SPM.

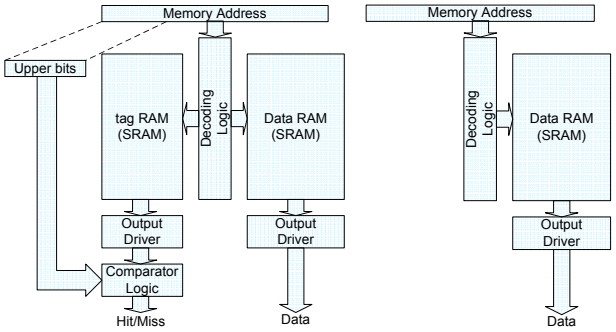
In this paper we present a novel architecture, containing a special hardware unit to manage dynamic copying of code segments from the main memory to the SPM. We describe the use of a specially created instruction which triggers copying from the main memory to the SPM. We further set forth heuristic algorithms which rapidly search for the sets of code segments to be copied into the SPM, and where to place the specially created instructions to maximize benefit. The whole system was evaluated using benchmarks from Mediabench [37] and UTDSP [38].

The rest of this paper is organized as follows: section II provides the motivation and the assumptions made for this work; section III describes previous works on SPM and presents the contributions of our work; section IV introduces our hardware strategy for using the SPM; section V defines the *concomitance* information; section VI formally describes the problem; section VII presents the heuristic algorithm for partitioning the application code; section VIII provides the experimental setup and results; and finally, section IX concludes this paper.

II. Motivation and Assumptions

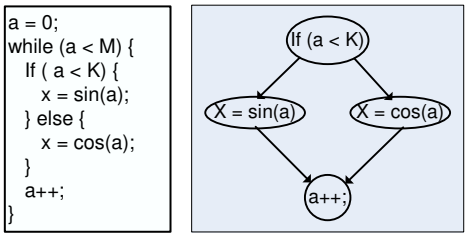
A. Motivation

An SPM is a memory comprised of only the data array (SRAM cells) and the decoding logic. This is illustrated in Figure 2(b). A



(a) Cache Architecture (b) SPM Architecture

Fig. 2. Architecture of Cache in comparison to SPM.



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

Fig. 3. Example Code

cache shown in Figure 2(a), in comparison, has data array (SRAM cells), decoding logic, tag array (SRAM cells), and comparator logic. Comparator logic checks if the existing content in cache matches the current CPU request and thus determines whether a cache hit or a cache miss has occurred. The access time and the access energy of individual components within the cache have been estimated using the tool CACTI [35]. CACTI results shows that for a cache system consisting of 8K bytes with associativities ranging from 1 to 32, the access time of the cache tag-RAM array is 38.9% longer on average compared to access time of the cache data-RAM array and the access energy of the data-RAM is on average 72.5% of the total cache access energy.

Each piece of data is placed into the cache according to its main memory location, identified by the tag. Thus, cache memory is transparent to the CPU. The SPM is used as a replacement of (level 1) instruction cache. The absence of the tag array within the SPM means that CPU has to know where a particular instruction reside within the SPM. To allow the CPU to locate data, SPM and the DRAM span a single memory address space, partitioned between them.

Selecting the correct code segments for placement onto 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 code segments to be placed in the SPM. Loop analysis has several drawbacks: (i) the structure and relationship of loops can be very complex and thus difficult to analyse; (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 of blocks of code matters, rather than the precise order of these instructions as provided by the loop analysis; see [9], [10], [11], [12]. Thus in this paper, instead of looking at the loop structure, we analyze temporal correlation of executions of blocks of code using statistical methods with a signal processing flavor. The trace is seen as a "signal" on which we perform a statistical, rather than a structural analysis.

We call the measure we introduce for temporal proximity of consecutive executions of the same block of the code *the self-concomitance* of the block, and we use it to decide if the block should be executed from the SPM or from the main memory. Similarly, the measure for temporal proximity of interleaved executions of two different blocks of the code we call *the concomitance*, and use it to decide whether two blocks should be placed in SPM simultaneously or they can overlap in the SPM.

Such analysis of the trace has proven to yield an algorithm for SPM placement with performance results that are not only superior to the loop analysis method, but is also much simpler, more efficient, and adaptive to different types of applications by changing only the distance function used to define the concomitance and self concomitance. Recently we found out that a related, but cruder and less general idea has been used for cache management in [13]. It uses a simple counting function rather than a decreasing function of distance, has no notion corresponding to our self-concomitance concept, and operates on coarser level of granularity (procedures rather than basic blocks).

B. Motivational Example for Concomitance

The benefits of using temporal proximity information is illustrated in the following example. Consider an "if-clause" shown in Figure 3(a), with the Control Flow Graph (CFG) shown in Figure 3(b). For $K = 50$ and $M = 100$ the profiling would determine that the first 50 execution of the "if-clause" will yield 50 executions of the block corresponding to $x = sin(a)$, while the next 50 execution of the "if-clause" will yield 50 executions of the block for $x = cos(a)$. Thus, executions of the block corresponding to $x = sin(a)$ will have large temporal distance to the executions of the block for $x = cos(a)$, and we will demonstrate that in such situation the concomitance of these two blocks will be small. Thus, blocks for $x = cos(a)$ and $x = sin(a)$ need not be placed into the SPM simultaneously, but can be placed to be overlapping with one another in the SPM, yet without degrading the performance because their execution does not overlap in time. Note that if only the frequency of execution was used, the CFG would only inform the algorithm that both blocks were executed 50 times, without providing crucial information that the pattern of execution is such that they can overlap on the scratch pad without any penalty. Note that should the "if-clause" be of the form $if(a\%2 == 0)$, executions of $x = sin(a)$ and $x = cos(a)$ would alternate. In such circumstances executions of the blocks corresponding to these two functions would be interleaved with short temporal distances apart. We will see that this makes their concomitance large, and the SPM placement algorithm would be informed that these two blocks must be placed into the SPM simultaneously and thus without overlap.

C. Assumptions

During the design and implementation of our methodology, the following assumptions are made.

- Size of the scratchpad memory or cache memory is only available in powers of two, and the size is passed to the algorithm. This is a reasonable assumption since the program is to be executed in an embedded system where the underlying hardware is known in advance. This assumption allows better optimization. If a number of processors with differing sizes of SPM have to be serviced, the SPM size can be made an adjustable parameter for the SPM management algorithm.

- Program size is larger than the SPM size. This is a valid assumption, because energy consumptions and cost constraints dictate

architectures with relatively small size SPM, compared to the size of a typical application.

- Size of the 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 rather small. However, if the basic block is too large for the SPM, it can be split into a smaller part that fits into the SPM and the remainder that is executed from the main memory.

- Each instruction is exclusively executed from either the SPM or the main memory. This ensures that it is never necessary to have duplicates of parts of the code or to alter branch destinations during the execution of the program.

- Program traces used for profiling provide an accurate depiction of the program behavior during its 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 [32].

- We do not consider higher level caches. In an embedded system, where frequently there is no cache at all, it is unlikely that more than a single level of cache is available. Moreover, having higher level caches would not reduce the effectiveness of the approach.

III. Related Work

In the past, use of SPM replacing cache memory has been shown to improve the performance and reduce energy consumption [8]. 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 [14], [15], [16], [17], [18], [19], [20], [21].

In 1997, Panda et al. [22] presented a scheme to statically manage an 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 rates by allowing data variables to be executed from SPM. Avissar et al. [23] 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. Experimental results shows that Avissar et al. were able to achieve 30% performance improvement over traditional direct-mapped data caches. By allowing part of the data stack to operate from SPM, performance improvement of 44.2% was obtained, compared to a system without SPM.

In 2001, Kandemir et al. introduced dynamic management of SPM for data memory [24]. Their memory architecture consisted of an SPM and a DRAM which was 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. Multiple optimization strategies were presented to partition data variables between SPM and DRAM. The best results were obtained from the hand optimized version. Their results shows a 29.4% improvement in performance compared to a traditional cache memory system. In addition, they also show that it is possible to obtain up to 10.2% performance improvement in using a dynamic management scheme for SPM compared to the static version. The work presented in [24] applied only to well-structured scientific and multimedia applications, with each loop nest needing independent analysis.

In 2003, Udayakumaran and Barua [25] improved upon [24] and presented a more general scheme for dynamic management of SPM as data memory. The method from [25] analyzes the whole program at once and aims to exploit locality throughout the whole program, instead of individual loop nesting. Reported reduction in execution time is by an average of 31.2% over the optimal static allocation scheme.

In 2002, Steinke et al. [26] presented statically managed SPM for both instruction and data. Their results shows on average 23% reduction in energy consumption over a cache solution.

In 2003, Angiolini et al. presented another SPM static management scheme for data memory and instruction memory that employs a polynomial time algorithm for partitioning data and instructions into DRAM and SPM [27]. Their results show energy improvement ranging from 39% to 84% over an unpartitioned SPM. In 2004, Angiolini et al. presented a different approach to static usage scheme for SPM [28] by mapping applications to existing hardware.

In [29], Steinke et al. presented a dynamic management scheme for instruction SPM. For each instruction to be copied into the SPM, they insert a load and a store instruction. The load instruction brings the instruction to be copied from the main memory to the processor, and a store instruction stores it back into SPM. Their algorithm for determining locations in the program for inserting load and store instructions is based on a solution of a Integer Linear Programming (ILP) problem that is obtained by using an ILP solver. They conducted experiments with small applications (bubble sort, heap sort, 'biquad' from DSP stone benchmark suit, etc.) and their results show on average a 29.9% energy reduction and a 25.2% performance improvement compared with a cache solution.

In 2004, Janapsatya et al. presented a hardware/software approach for dynamic management scheme of SPM [30]. Their approach introduces a specially designed hardware component to manage the copying of instructions from the main memory into the SPM. They utilize instructions execution frequency information to model applications as graph and perform graph partitioning to determine locations within the program for initiating the copying process.

A. Contributions

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

- we design a novel architecture to perform dynamic management of instruction code between SPM and main memory;
- we replace 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 without an overlap;
- we develop a novel graph partitioning algorithm for splitting the instruction code into segments that will either be copied into the SPM or executed directly from the main memory, in a way that reduces the 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.

IV. System Architecture

A. Hardware Implementation

The proposed methodology modifies the processor by adding an SPM controller, which is responsible for copying instructions from DRAM to SPM and stalling the CPU whenever copying is in progress. Figure 4 shows the block diagram of the SPM controller, SPM, DRAM, and the CPU. A new instruction called SMI (Scratchpad Managing Instruction) is implemented. The SMI is used to activate the SPM controller; SMIs are inserted within the program and are executed whenever the CPU encounters them. The cost of the CPU stalling while the SPM is being copied is identical to servicing a cache miss; and the number of times the SMI was

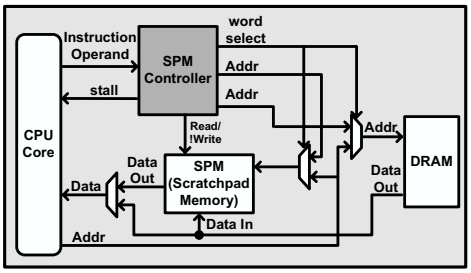


Fig. 4. SPM system architecture

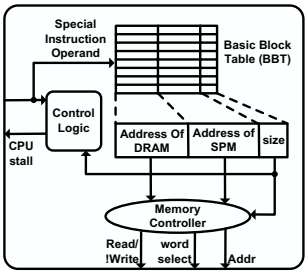


Fig. 5. Architecture of the SPM controller

executed provides useful information for determining the application execution time. Further performance improvement is possible, if copying can be performed a few cycles before the instructions are to be executed assuming there is no conflict in the SPM location between the instructions to be executed and the instructions to be copied.

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 blocks into the SPM. Content of the BBT is filled by the algorithm shown in Figure 6.

With the addition of SPM, an example of program execution is as follows: at the start of a program, the first instruction is always fetched from DRAM. When an SMI is executed, the CPU will activate the SPM controller. The SPM controller will then stall the CPU, read information from the BBT and provide addresses to DRAM and SPM for copying instructions from DRAM into SPM. After copying is completed, the SPM controller will release the CPU, which will then continue to fetch and execute instructions from DRAM. Whenever a branching instruction leads program execution to the SPM, the CPU continues to fetch instructions from the SPM.

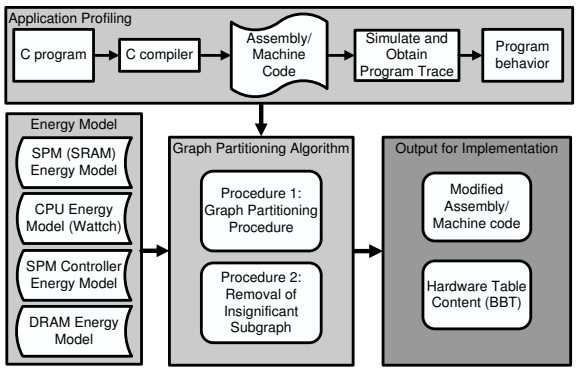


Fig. 6. Methodology for Implementing the SPM

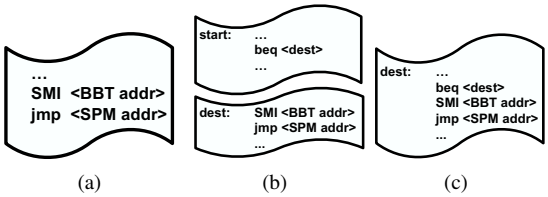


Fig. 7. Condition for insertion of SMI into programs.

Figure 6 shows the steps for implementing the proposed methodology. The methodology operates on the assembly/machine code level. Profiling and tracing (using modified simplescalar/Pisa 3.0d [31] to output instruction trace into a gzip file) is performed on the assembly/machine code to obtain an accurate representation of the program behavior. Program trace was generated using SimpleScalar 3.0d out-of-order execution. To ensure proper order of execution of SMI in out-of-order machine, the instructions to be copied should be made dependent on the SMI. This is similar to a load operation followed by register operation upon the loaded data. Program behavior is captured by extracting the concomitance information as explained below. The SPM placement algorithm uses the concomitance information to determine appropriate locations for inserting SMI within the program, and to construct table (BBT) that specifies the positions where the blocks will be copied.

B. 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 a 16 bit operand, it is possible to have up to 65536 entries in the BBT, allowing 65536 SMIs to be inserted into a program.

The code can be modified by insertion of an SMI in the following three cases:

(1) An SMI can be inserted just before an unconditional branch (Figure 7(a)); in this case the destination address of the unconditional branch is altered to point to the correct address within the SPM.

(2) An SMI can be inserted to be the new destination of a conditional branch (Figure 7(b)); if the branch tests positive, such an SMI will copy the basic block to be executed from the appropriate location in the SPM; an extra jump instruction is added immediately after the SMI, to transfer the program execution to that SPM location.

(3) An SMI instruction can be inserted just after a conditional branch instruction (Figure 7(c)). In this case, if the conditional branch tests negative, the SMI will copy the basic block that is to be executed following the branch instruction. Again, an extra jump instruction is added immediately after the SMI to transfer program execution to the SPM.

An extra branch instruction may also be needed 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 6, outputs the modified software including the addition of any extra branching instructions.

V. 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. such that if u,v are real numbers and $u \leq v$ then $W(u) \geq W(v)$.

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:

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

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 a 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 are taken over executions with a 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 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 $\sigma(b,T)$.

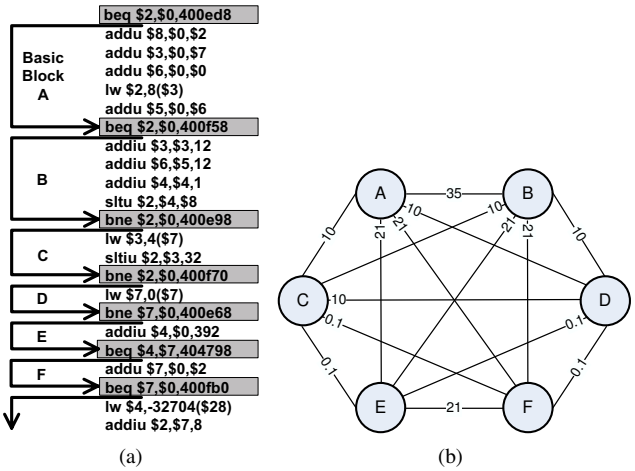


Fig. 8. An example of the concomitance graph.

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.
- *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; 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 in instructions.

VI. Problem Description

To copy the important code segments into SPM, SMIs are inserted into strategic locations within the program. To determine strategic locations for insertion of SMIs, we transformed the problem into a graph partitioning problem.

Given any program, it can be represented as a graph as follows. The vertices represent basic blocks belonging to the set S of basic blocks having large self-concomitance. The weight of each vertex represents the number of instructions within the basic block (size of the basic block) and the weight of each edge represents the concomitance value between the blocks joined by the edge. An illustration of the basic block is given in Figure 8(a) and its corresponding concomitance graph is shown in Figure 8(b).

```
Construct a graph containing all the vertices in S;
Sort list of edges in ascending edge weight value.
Repeat until total weight of each subgraph is less than Z.
    Eliminate set of edges that have weight value 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 Z,{
            Form a list of its edges preserving the ordering
            from the initial list.
        }
    }
}
```

Fig. 9. Graph partitioning procedure

The problem is to find vertices within the graph (i.e. the basic blocks of instructions) where one should insert the custom instruction, SMI.

VII. Code Partitioning Algorithm

A. Graph Partitioning Problem

Consider the following *Graph Partitioning 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 weight $\tau(a,b,T)$. (the concomitance value);*
3) *a constant Z (size of SPM)*

Find a partition of the graph, such that each subgraph has a total vertex weight less than or equal to Z, and the sum of edge weights of all the edges connecting these subgraphs is minimal.

It is easy to see that the "Minimum Cut into Bounded Sets Problem" [33] is P-time Turing reducible to our graph partitioning problem; thus we cannot expect to find a polynomial time algorithm, and we must use a heuristic approximation algorithm to work on real applications. We implemented an algorithm consisting of two procedures: Graph Partitioning Procedure and Removal of Insignificant Subgraphs Procedure, as shown on Figure 9.

We start with the set S of basic blocks whose self-concomitance is larger than the chosen threshold, and consider a graph whose vertices are all elements of S. All vertices of this graph are initially pairwise connected, thus forming a complete graph (a clique). The edges have weights that are equal to the concomitance between their two vertices. We now order the set of edges E according to their weights in an ascending way; thus, the edge at the top of the list is one of the edges with the lowest concomitance value. Notice that, since we start with a complete graph, $|E| = \frac{|S|\cdot(|S|-1)}{2}$. We eliminate all the edges of the graph which have such a minimal weight value. This usually induces a partition of the graph into several connected subgraphs; the total weight of the vertices for each of these connected subgraphs is calculated and compared to Z. We then apply the same procedure to all connected subgraphs whose total vertex weight is larger than Z, and the procedure stops when all of the connected subgraphs have a total vertex weight smaller or equal to Z. It is easy to see that the complexity of the graph partitioning procedure is $\Theta(|S|\cdot|E|) = \Theta(|S|^3)$.

After we have produced the subgraphs using the concomitance, we now consider the CFG graph with edges having weights equal to the total number of traversals of the edge during execution.

```
For each subgraph G {
    Calculate energy cost E_DRAM(G) of executing G from DRAM,
    Calculate energy cost E_SPM(G) of executing G from SPM,
    if E_SPM(G) < E_DRAM(G)
        classify G to be executed from SPM,
    else
        classify G to be executed from DRAM.
}
For each edge e in IE(G) {
    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 G;
        if no, then insert SMI between the vertices
        connected by the edge e.
    }
}
```

Fig. 10. Removal of insignificant subgraph procedure

First of all, there can be subgraphs that are not worthwhile to copy and execute from SPM. For example, the graph partitioning procedure can produce results such as the one shown in Figure 11. 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.

B. Removal of Insignificant Subgraphs Procedure

Such a heuristic procedure is shown in Figure 10. It starts by calculating the energy cost of executing each subgraph G from DRAM and the energy cost of executing such subgraph from SPM. Energy cost of executing G from DRAM, $E_{DRAM}(G)$, is calculated using

$$E_{DRAM}(G) = DRAM_{acc} * \sum_{v \in G} (W(v) * f(v)) \quad (1)$$

where $DRAM_{acc}$ is the energy cost of a single DRAM access per instruction, $W(v)$ is the number of instructions in the basic block corresponding to the vertex v, and $f(v)$ is the total number of executions of the basic block corresponding to the vertex v. Equation 2 shows the energy cost $E_{SPM}(G)$ of executing instructions within the subgraph G from SPM, including the cost of accessing the DRAM once and copying to SPM.

$$E_{SPM}(G) = SPM_{acc} * \sum_{v \in G} (W(v) * f(v)) + E_{special}(G) \quad (2)$$

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

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

$IE(G)$ is the set of all edges that are incoming to the subgraph G from all other subgraphs; $E_{special}(G)$ is the energy cost of executing the special instruction including reading from DRAM and copying to SPM, and E_{branch} is the cost 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 for an edge e, constant $q(e)$ is equal to 1, else $q(e)$ is 0.

The energy calculation is used to classify which subgraphs are to be executed from SPM and which subgraphs are to be executed

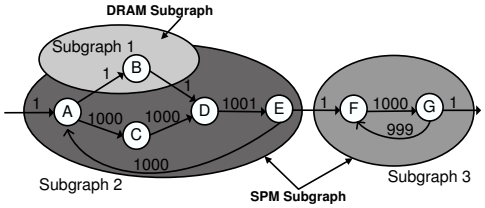


Fig. 11. Subgraph Representation

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 I
SIMPLESCALAR CONFIGURATION.

from DRAM; only subgraphs G with $E_{SPM}(G) < E_{DRAM}(G)$ will be executed from SPM. We call such a subgraph an SPM-subgraph G.

The rest of the procedure inserts the SMI as follows. For every SPM-subgraph, it will examine all incoming edges to this subgraph $IE(G)$. We determine if such an edge is included in a path either from another SPM-subgraph G_1, or from the start of the program, and only in such cases an SMI is inserted. This means that if all paths including this edge emanate from the same SPM-subgraph G, an SMI need not be inserted. The complexity of this procedure is $\Theta(|S|^2)$, where S is the number of vertices in the graph.

For example, in the CFG shown in Figure 11, there are three subgraphs created by the graph partitioning procedure. Subgraph 1 consists 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. In this way we avoid unnecessary replacement of instructions that are already in the SPM.

The algorithm described in this section assumed the use of the concomitance value for the partitioning of the graph. Similar algorithm can be used with frequency (as in [30]) instead of concomitance, and the results are compared in Section VIII.

VIII. EXPERIMENTAL RESULTS

A. Experimental Setup

We simulated a number of benchmarks using simplescalar/PISA 3.0d [31], to obtain memory access statistics. Power figures for the CPU were calculated using Wattch [34] (for a $0.18\mu m$ process.). CACTI 3.2 [35] was used as the energy model for the cache memory. The energy model for the scratchpad memory was extracted from CACTI as in [8]. The DRAM power figures were taken from IBM embedded DRAM SA-27E [36]. The configuration for the simulated CPU is as shown in Table I.

All benchmarks were obtained from the Mediabench suite [37] except for the benchmark histogram which were from UTDSP test suite [38]. The total number of instructions executed in each benchmark is tabulated in Table II.

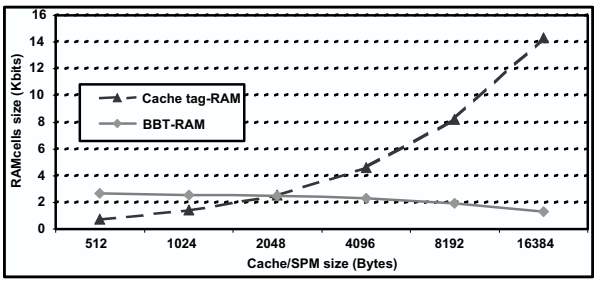


Fig. 12. Cache tag-RAM size compared to BBT size in bits.

B. Area Cost for Inclusion of SMI Hardware Controller

In cache, the tag array memory cells keep record of the entries within the cache data array. While for the SPM system proposed here, the BBT keeps a record of where each set of code segment(s) are to be placed within the SPM. 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 2(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 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 12 shows comparison between the number of bits in a cache tag RAM cells and the average size of the BBT for different size cache/SPM with a main memory size of 2M bytes. The average size of the BBT is calculated from all the benchmarks shown in Table II. From Figure 12, it can be seen that cache tag RAMcells grows linearly as the cache size grows exponentially, but the BBT size decreases as the SPM grows exponentially. BBT size decreases as SPM size increases because with a larger SPM, fewer SMIs are needed.

Table III showed a comparison of the total number of memory accesses. Performance and energy results measured from the experiments are shown in Table IV and Table V, and cost of adding copy locations within the program is shown in Table II.

In Table II, 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 using the concomitance method, column 6 and 7 present the results from using frequency as the edge value ([30]). Comparing figures in Table II, column 5 and column 7 shows that the number of instructions to be copied into the SPM is significantly reduced using the concomitance method compared to the frequency method.

An SPM controller with BBT size of 128 entries (Total size = 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: $ClockFreq. = 500MHz$; $Voltage 1.8V$; 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

Application	Prog. size (insn.)	Total no. of insn. exec.	*Concomitance* method		Frequency method	
			SMI insns inserted	Avg. no. of insn. copied into SPM	SMI insns inserted	Avg. no. of insn. copied into SPM
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
pegwitenc	20562	13483894	5.6	29795	81.6	506801
pegwitdec	20562	18941701	4.2	15793	120.0	1131996
pegwitkey	20562	33700290	3.2	2085	113.1	601407
histogram	12620	21328862	7	2829	47	2943470

TABLE II
COST OF ADDING AND EXECUTING COPY INSTRUCTIONS.

of 159mW, obtained from CACTI [35], for a $0.18\mu m$ 64 bytes of direct map cache. The cache power would progressively increase for greater ways of associativity. In addition, we also synthesized the SPM controller using Synopsys tools [39], for a $0.18\mu m$ process. The result shows that the SPM controller is approximately 1108 gates in size. Comparing the size of the SPM controller with a synthesized version of the simplescalar CPU (obtained from [40], the size of the synthesized CPU without the cache memories is approximately 82200 gates.) shows that the SPM controller is approximately 1.35% of the total CPU size.

C. Performance Cost to Accommodate The Execution of SMI

Table III compares the total number of memory accesses of a cache system, the system with scratchpad using the frequency method for capturing the program behavior ([30]), and the system with scratchpad using the concomitance metric as a method for capturing the program behavior. Column 1 in Table III 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 using the frequency optimization method; column 9 shows the total DRAM accesses obtained from using the concomitance optimization method; column 10 shows the percentage improvement of the concomitance method over the average values of the results of the cache system (in columns 3 to column 7); and column 11 compares the percentage improvement of the concomitance method over the frequency method. In column 11, it is shown that the concomitance method reduces the total number of DRAM accesses by an average of 52.94% compared to the frequency method. When comparison is made with a cache based system (column 3 to 7), it is shown that the total number of DRAM accesses for the SPM system can be larger 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 translate to worse energy consumption and worse performance compared to a cache system (such cases are highlighted in **bold** in Tables III, IV, and V.) 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 an SPM system is comprised of both the number of instructions to be executed from DRAM and the total number of instructions 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.

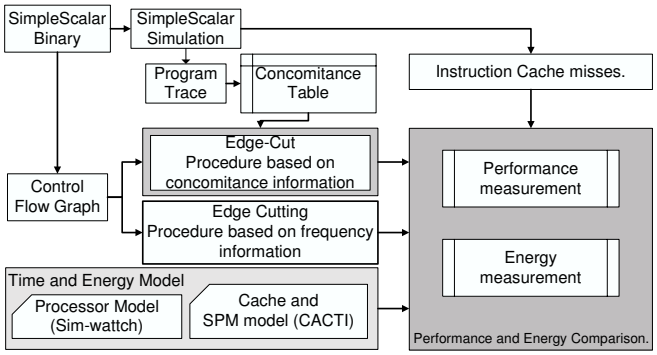


Fig. 13. Experimental Setup.

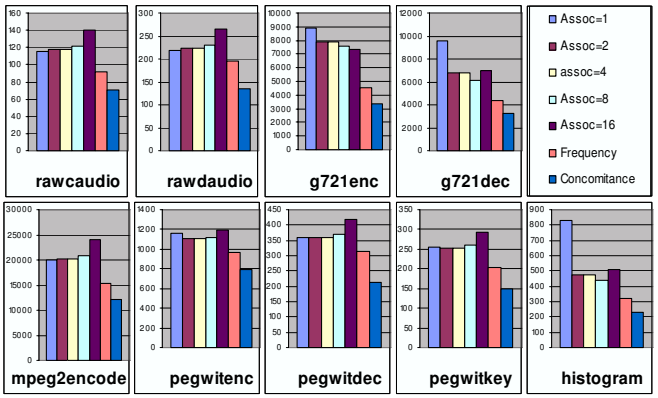


Fig. 14. Energy Comparison for 8K bytes SRAM. (y-axes are energy in mJ)

D. Analysis of Results

The experimental setup for measuring the performance and energy consumption is shown in Figure 13. 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.

Total access time of the cache architecture system is calculated using equation 4

$$cache_{access\_time}(ns) = (cache_{hit} + cache_{miss}) * cache_{time}(ns) + cache_{miss} * DRAM_{time}(ns) \quad (4)$$

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

SRAM size	Total cache misses assoc = 1	2	4	8	16	DRAM acc. (freq.)	DRAM acc. (*conc.*)	% imp. of *conc.* over cache (avg.)	% imp. of *conc.* over freq.
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
pegwitenc									
1024	1.0E+7	**9.9E+6**	9.8E+6	**9.8E+6**	9.6E+6	9.4E+6	8.6E+6	12.4	8.6
2048	9.3E+6	**8.7E+6**	8.6E+6	**8.6E+6**	8.6E+6	9.0E+6	8.0E+6	8.4	10.7
4096	7.3E+6	**8.3E+6**	8.3E+6	**8.4E+6**	8.4E+6	9.0E+6	8.0E+6	1.6	10.7
8192	3.2E+6	**3.5E+6**	3.0E+6	**3.0E+6**	3.0E+6	3.4E+6	2.9E+6	6.8	13.7
16384	3.6E+5	**6.4E+4**	4.8E+4	**6.0E+4**	6.6E+4	3.6E+5	6.7E+4	4.8	81.2
pegwitdec									
1024	6.3E+6	**6.2E+6**	6.1E+6	**6.2E+6**	6.0E+6	5.9E+6	5.5E+6	10.8	7.4
2048	5.9E+6	**5.6E+6**	5.5E+6	**5.5E+6**	5.5E+6	5.7E+6	5.2E+6	6.9	8.9
4096	4.6E+6	**5.4E+6**	5.4E+6	**5.4E+6**	5.4E+6	5.7E+6	5.2E+6	0.1	8.9
8192	1.8E+5	**1.9E+5**	1.4E+5	**1.4E+5**	1.2E+5	3.7E+5	8.0E+4	46.8	78.6
16384	7.1E+4	**6.3E+4**	6.0E+4	**7.6E+4**	8.5E+4	2.5E+5	6.6E+4	6.2	73.5
pegwitkey									
1024	1.2E+6	**1.1E+6**	9.9E+5	**1.0E+6**	8.8E+5	7.7E+5	3.9E+5	61.8	49.8
2048	7.5E+5	**4.4E+5**	3.8E+5	**3.8E+5**	3.8E+5	5.3E+5	7.9E+4	81.8	85.0
4096	2.7E+5	**2.6E+5**	2.7E+5	**2.9E+5**	2.9E+5	5.3E+5	7.9E+4	71.2	85.0
8192	1.2E+5	**1.3E+5**	8.8E+4	**8.1E+4**	6.3E+4	9.7E+4	3.9E+4	56.6	59.5
16384	3.9E+4	**3.1E+4**	2.6E+4	**3.2E+4**	3.5E+4	9.7E+4	3.9E+4	**-22.9**	59.5
histogram									
1024	1.6E+7	**1.7E+7**	1.8E+7	**1.8E+7**	1.9E+7	1.9E+7	1.7E+7	5.7	14.6
2048	1.3E+7	**1.3E+7**	1.4E+7	**1.4E+7**	1.4E+7	1.4E+7	9.0E+6	34.1	37.4
4096	8.2E+6	**7.1E+6**	4.9E+6	**3.8E+6**	3.7E+6	1.4E+7	3.4E+6	32.0	74.8
8192	5.7E+6	**1.7E+6**	2.6E+5	**1.5E+4**	3.5E+3	7.8E+4	9.2E+3	34.6	88.2
16384	4.2E+6	**4.5E+5**	3.3E+3	**3.0E+3**	2.9E+3	8.5E+3	2.8E+3	44.0	66.3

TABLE III
TOTAL MEMORY ACCESS COMPARISON.(CACHE RESULT SHOWS THE TOTAL CACHE MISSES. DRAM ACCESS IS THE TOTAL NUMBER OF INSTRUCTIONS EXECUTED FROM DRAM PLUS THE NUMBER OF INSTRUCTIONS COPIED FROM DRAM TO THE SPM.)

SRAM size	Cache System's Energy (mJ) assoc = 1	2	4	8	16	SPM (mJ) (freq.)	SPM (mJ) (*conc.*)	% imp. of *conc.* over cache (avg.)	% imp. of *conc.* over freq.
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
pegwitenc									
1024	2161	**2141**	2141	**2200**	2242	1780	1621	25.5	8.9
2048	1964	**1949**	1949	**2014**	2082	1712	1543	22.5	9.9
4096	1929	**1947**	1947	**1977**	2088	1730	1554	21.4	10.2
8192	1157	**1105**	1105	**1116**	1192	970	795	29.9	18.0
16384	717	**711**	711	**752**	838	553	381	48.7	31.1
pegwitdec									
1024	1312	**1310**	1310	**1337**	1365	1106	1016	23.4	8.1
2048	1215	**1211**	1211	**1244**	1286	1071	978	20.7	8.7
4096	1201	**1208**	1208	**1231**	1288	1081	984	19.8	9.0
8192	361	**361**	361	**370**	419	313	213	42.9	31.9
16384	411	**407**	407	**431**	483	322	224	47.3	30.3
pegwitkey									
1024	379	**374**	374	**394**	407	240	180	53.2	24.9
2048	282	**276**	276	**300**	330	205	141	51.6	31.2
4096	264	**269**	269	**287**	328	215	147	47.8	31.6
8192	256	**254**	254	**260**	294	205	149	43.4	27.4
16384	289	**286**	286	**303**	340	219	158	**47.2**	27.8
histogram									
1024	3468	**3585**	3585	**3656**	3793	3061	2740	24.2	10.5
2048	3050	**3122**	3122	**3120**	3176	2094	1580	49.3	24.5
4096	1921	**1708**	1708	**1591**	1442	1624	735	55.7	54.8
8192	830	**474**	474	**441**	507	322	229	55.8	29.0
16384	607	**504**	504	**532**	596	339	243	55.4	28.3

TABLE IV
SYSTEM'S ENERGY COMPARISON.

instruction from the cache, and $DRAM_{time}$ is the amount of time spent to access one DRAM instruction.

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.

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

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

Energy consumption comparison is shown in Table IV. The cache energy consumption is calculated using equation 7,

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

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

SRAM size	Cache System Execution Time (ms) assoc = 1	**2**	4	**8**	16	SPM (ms) (freq.)	SPM (ms) (*conc.*)	% imp. of *conc.* over cache (avg.)	freq.
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
pegwitenc									
1024	169.0	**166.6**	166.6	**169.7**	169.9	136.6	128.8	23.5	5.7
2048	153.5	**151.5**	151.5	**155.1**	157.3	131.4	122.6	20.3	6.7
4096	150.7	**151.3**	151.3	**152.2**	157.8	132.8	123.4	19.1	7.1
8192	90.1	**85.2**	85.2	**84.4**	87.4	73.1	62.8	27.3	14.1
16384	55.6	**54.3**	54.3	**56.0**	59.7	40.3	29.6	47.0	26.5
pegwitdec									
1024	102.6	**102.0**	102.0	**103.3**	103.7	84.9	80.8	21.3	4.8
2048	95.0	**94.2**	94.2	**95.9**	97.5	82.2	77.7	18.5	5.4
4096	93.8	**93.9**	93.9	**94.9**	97.6	83.0	78.2	17.5	5.8
8192	27.9	**27.4**	27.4	**27.2**	29.4	22.7	16.5	40.6	27.1
16384	31.8	**31.0**	31.0	**32.1**	34.4	23.3	17.4	45.6	25.3
pegwitkey									
1024	29.5	**28.8**	28.8	**29.7**	29.6	17.8	14.1	51.8	20.6
2048	21.9	**21.1**	21.1	**22.4**	23.5	15.1	11.0	50.0	27.1
4096	20.5	**20.6**	20.6	**21.3**	23.3	15.8	11.4	46.0	27.6
8192	19.8	**19.3**	19.3	**19.2**	20.6	15.0	11.6	41.0	22.6
16384	22.4	**21.9**	21.9	**22.6**	24.2	16.1	12.3	**45.3**	23.2
histogram									
1024	271.6	**280.3**	280.3	**284.8**	293.3	238.3	218.5	22.5	8.3
2048	238.9	**244.0**	244.0	**242.7**	244.9	162.9	125.8	48.2	22.8
4096	150.2	**132.9**	132.9	**122.7**	108.8	127.0	58.2	54.6	54.2
8192	64.6	**36.1**	36.1	**32.4**	35.4	22.9	17.7	54.0	22.8
16384	47.1	**38.4**	38.4	**39.5**	42.4	24.1	18.8	54.0	22.1

TABLE V
SYSTEM'S PERFORMANCE COMPARISON.

The SPM energy is calculated using equation 8,

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

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 SMIs are executed, E_{branch} is the energy cost for any additional branch instructions, and J is the total number of times any additional branch instructions are executed.

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

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

Results of the percentage energy improvement is shown in Table IV.

Table IV shows energy comparison of: (i) a cache system; (ii) a scratchpad system using the frequency method [30] and; (iii) a scratchpad system using the concomitance method. The table structure is identical to Table III except the comparison is now for energy. Figure 14 shows the energy improvement comparison between the cache system, the frequency SPM allocation procedure [30], and the concomitance SPM allocation method for all the benchmarks. The energy comparison shows that the concomitance method almost always performs better compared to the cache system, and superior results are seen when compared with results of the frequency method. On average, the energy consumption by utilizing the concomitance method is 41.9% better than cache system energy, and 27.1% better than the frequency method. In particular the concomitance method is superior in cases where negative improvements over cache were shown using the frequency method.

Performance results are shown in Table V. The structure of the table is identical to Table IV; column 3 to column 7 shows the execution time of a cache based system; column 8 shows the performance results of the frequency method; and column 9 shows the performance measurement results of the concomitance method; column 10 shows the ***average*** performance improvement of the concomitance method over cache system; and column 11 shows performance improvement over the frequency method. The concomitance method improves the execution time by 40.0% compared to cache and 23.6% compared to the frequency method.

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 knowing which multi-cycle instructions were executed. We perform comparisons between cache execution times obtained by Simplescalar simulation, to cache memory access times obtained from equation 4. We found that on average 9.5% error is seen between the two values with a maximum error of 17%.

Thus it is clear from the results that the method is feasible for utilizing a dynamic SPM allocation scheme. We show that the number of copy instructions inserted are far fewer than the existing methods described in [29] and [30]. In addition, for the applications shown here, we show performance improvement and energy savings over conventional cache based systems.

IX. CONCLUSIONS

We have presented a method to lower energy consumption and improve performance of embedded systems. We presented a methodology for selecting appropriate code segments to be executed from the SPM. The methodology determined the code segments that are to be stored within the SPM by using the concomitance information. 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, to activate the hardware SPM controller. Experimental results show that our SPM scheme can lower energy consumption by an average of 41.9% compared to traditional cache architecture, and performance is improved by an average of 40.0%.

REFERENCES

[1] K. Lahiri, S. Dey, and A. Raghunathan, "Communication Architecture Based Power Management for Battery Efficient System Design," *Design Automation Conference. 39th Annual Proceedings of,* 2002.

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

[3] F. Sun, S. Ravi, A. Raghunathan, and N. K. Jha, "Custom-Instruction Synthesis for Extensible-Processor Platforms," *Computer Aided Design. IEEE Transactions on,* vol.23, no.2, pp.216-228, 2004.

[4] N. Cheung, S. Parameswaran, and J. Henkel, "A quantitative study and estimation models for extensible instructions in embedded processors," *International Conference on Computer Aided Design. Proceedings of,* 2004.

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

[6] J. Kin, G. Munish, and W. H. Mangione-Smith, "The Filter Cache: An Energy Efficient Memory Structure," *Microarchitecture. 30th Annual IEEE conference on,* 1997.

[7] J. Montanaro et al., "A 160MHz, 32b, 0.5W CMOS RISC microprocessor," *Solid State Circuit. IEEE Journal of,* vol.31, no.11, pp. 1703-1714, 1996.

[8] R. Banakar et al., "Scratchpad Memory: A Design Alternative for Cache On-chip Memory in Embedded Systems," *Conference on Hardware/Software Codesign. Proceedings of,* 2002.

[9] G. Ramalingam, "On Loops, Dominators, and Dominance Frontier," *Programming Language Design and Implementation. Conference on,* pp.233-241, 2000.

[10] P. Havlak, "Nesting of Reducible and Irreducible Loops," *Programming Languages and Systems. ACM Transactions on,* vol.19, no.4, 1997.

[11] V. C. Sreedhar, G. R. Gao, and Y. F. Lee, "Identifying Loops using DJ Graphs," *Programming Languages and Systems. ACM Transactions on,* vol.18, no.6, 1996.

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

[13] 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.

[14] S. Parameswaran and J. Henkel, "I-CoPES: fast instruction code placement for embedded systems to improve performance and energy efficiency," *International Conference on Computer Aided Design. Proceedings of,* 2001.

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

[16] S. McFarling, "Program optimization for instruction caches," *Architectural Support for Programming Language and Operating Systems. Proceedings of the 3rd,* 1989.

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

[18] P. R. Panda, N. D. Dutt, and A. Nicolau, "Memory Organization for Improved Data Cache Performance in Embedded Processors, *International Symposium on System Synthesis. Proceedings of the 9th,*, 1996.

[19] P. R. Panda, H. Nakamura, N. D. Dutt, and A. Nicolau, "A Data Alignment Technique for Improving Cache Performance," *International Conference on Computer Design. Proceedings of,*, 1997.

[20] H. Tomiyama and H. Yasuura, "Optimal code Placement of Embedded Software for Instruction Cache," *The European Conference European Exhibition on Design Automation Test Conference,* 1996.

[21] S. Bartolini and C. A. Prete, "A cache-aware program transformation technique suitable for embedded systems," *Information and Software Technology,* vol.44, no.13, 2002.

[22] P. R. Panda, N. D. Dutt, and A. Nicolau, "Efficient Utilization of Scratch-Pad Memory in Embedded Processor Applications," *European Design and Test Conference, Proceedings of,* 1997.

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

[24] M. Kandemir and A. Choudhary, "Compiler-Directed Scratch Pad Memory Hierarchy Design and Management," *Design Automation Conference. Proceedings of 39th,* 2002.

[25] S. Udayakumaran and R. Barua, "Compiler-Decided Dynamic Memory Allocation for Scratch-Pac Based Embedded Systems," *International Conference on Compilers, Architecture, and Synthesis for Embedded Systems. Proceedings of the,* 2003.

[26] S. Steinke, L. Wehmeyer, B. Lee, and P. Marwedel, "Assigning Program and Data Objects to Scratchpad for Energy Reduction," *Conference on Design, Automation, and Test in Europe. Proceedings of the,* 2002.

[27] F. Angiolini, L. Benini, and A. Caprara, "Polynomial-Time Algorithm for On-Chip Scratchpad Memory Partitioning," *International Conference on Compilers, Architecture, and Synthesis for Embedded Systems. Proceedings of the,* 2003.

[28] F. Angiolini et al., "A Post-Compiler Approach to Scratchpad Mapping of Code," *International Conference on Compilers, Architecture, and Synthesis for Embedded Systems. Proceedings of the,* 2004.

[29] S. Steinke et al., "Reducing Energy Consumption by Dynamic Copying of Instructions onto Onchip Memory," *International Symposium on System Synthesis. Proceedings of the 15th,* 2002.

[30] A. Janapsatya, A. Ignjatovic, and S. Parameswaran, "Hardware/Software Managed Scratchpad Memory for Embedded System", *International Conference on Computer Aided Design. Proceedings of,* 2004.

[31] D. Burger and T. M. Austin, "The SimpleScalar Tool Set, Version 2.0," *TR-CS-1342 (http://www.simplescalar.com),* University of Wisconsin-madison, June 1997.

[32] 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.

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

[34] D. Brooks, V. Tiwari, and M. Martonosi, "Wattch: A Framework for Architectural-Level Power Analysis and Optimizations," *International Symposium on Computer Architecture. Proceedings of the 27th,* 2000.

[35] 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.

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

[37] C. Lee et al., "MediaBench: A Tool for Evaluating Multimedia and Communications Systems," *Microarchitecture. 30th Annual IEEE conference on,* 1997.

[38] C. Lee and M. Stoodley, "UTDSP Benchmark Suite," *http://www.eecg.toronto.edu/ corinna/DSP/infrastructure/UTDSP.html,* 1992.

[39] Synopsys Inc., "Synopsys - Electronic Design Automation Solutions and Services," *http://www.synopsys.com*, 2005.

[40] J. Peddersen, S. L. Shee, A. Janapsatya, and S. Parameswaran, "Rapid Embedded Hardware/Software System Generation," *International Conference on VLSI Design. Proceedings of the 18th,* 2005.

Andhi Janapsatya (S'00-M'06) received his B.E. (Honors) degree in electrical engineering from The University of Queensland in Brisbane, Australia, in 2001. He is currently pursuing his Ph.D. at The University of New South Wales in Sydney, Australia (submitted Oct 2005). His research interest include low power design, application specific hardware design, and energy and performance optimization of application specific systems.

Aleksandar Ignjatovic̀ received Bachelor's and Master's degrees in Mathematics at the University of Belgrade, former Yugoslavia, and in 1990 Ph.D. in Mathematical Logic at the University of California at Berkeley. He was an Assistant Professor of Mathematical Logic and Philosophy at the Carnegie Mellon University before starting *Kromos Technology Inc*. Presently he is a senior lecturer at the School of Computer Science and Engineering at the University of New South Wales in Sydney. His interests include applications of Mathematical Logic to Complexity Theory, Approximation Theory, Algorithms and Embedded Systems Design.

Sri Parameswaran is an Associate Professor in the School of Computer Science and Engineering. He also serves as the Program Director for Computer Engineering at the University of New South Wales. Sri has received the Faculty of Engineering Teaching Staff Excellence Award in 2005. His research interests are in System Level Synthesis, Low power systems, High Level Systems and Network on Chips. He has served on the Program Committees of numerous International Conferences, such as Design and Test in Europe (DATE), the International Conference on Computer Aided Design (ICCAD), the International Conference on Hardware/Software Codesign and System Synthesis (CODES-ISSS), Asia South Pacific Design Automation Conference (ASP-DAC) and the International Conference on Compilers, Architectures and Synthesis for Embedded Systems (CASES). He has received best paper nominations from the International Conference on VLSI Design and the Asia South Pacific Design Automation Conference.