Embedded Intelligent SRAM

Prabhat Jain G. Edward Suh Srinivas Devadas
Laboratory for Computer Science
Massachusetts Institute of Technology
200 Technology Square
Cambridge, MA 02139
{prabhat, suh, devadas}@mit.edu

ABSTRACT

Many embedded systems use a simple pipelined RISC processor for computation and an on-chip SRAM for data storage. We present an enhancement called Intelligent SRAM (ISRAM) that consists of a small computation unit with an accumulator that is placed near the on-chip SRAM. The computation unit can perform operations on two words from the same SRAM row or on one word from the SRAM and the other from the accumulator. This ISRAM enhancement requires only a few additional instructions to support the computation unit. We present a computation partitioning algorithm that assigns the computations to the processor or to the new computation unit for a given data flow graph of a program. Performance improvement results from the reduction in the number of accesses to the SRAM, the number of instructions, and the number of pipeline stalls compared to the same operations in the processor. Experimental results on various benchmarks show up to 1.48× speedup with our enhancement.

Categories and Subject Descriptors

B.3.1 [**Semiconductor Memories**]: Static memory (SRAM); C.1.3 [**Processor Architectures**]: Pipeline processors; D.3.4 [**Programming Languages**]: Code generation.

General Terms

Algorithms, Design, Management, Performance.

Keywords

Embedded, Intelligent, SRAM, Computation Partitioning.

1. INTRODUCTION

Embedded systems are used in a variety of applications. Some examples of embedded systems are hand-held devices, automotive systems, and portable communication devices. Unlike general-purpose systems, embedded systems need to be cheap and have low power dissipation. Thus, complex superscalar processors are inappropriate for use in most embedded systems. As a result, many embedded systems use simple pipelined RISC processors such as the ARM9TDMI [1] and others use micro-controllers or DSP processors.

We propose an enhancement to pipelined embedded RISC processors, which places a small computation unit with an accumulator near the on-chip SRAM. We call this additional computation unit `S-ALU`, and the combination of the SRAM and `S-ALU` ISRAM. The computation unit operates under the control of the processor through the use of a few additional instructions. Additional data transfer instructions allow data movement among the processor register file, accumulator, and SRAM. This enhancement requires only minor modifications to the processor pipeline and a few additional instructions.

One of the key features of `S-ALU` is that it can operate on two words from the same row of the SRAM. The on-chip SRAMs are organized as rows of words that are accessed at the same time. The rows may range from 32 bits to 128 bits. When a word is to be read from the SRAM, the whole row containing the word is read and the desired word is selected from the row of words. In our enhancement, the additional computation unit can read two words from the same row and perform the computation with them in one instruction, which saves one SRAM access. Because the entire row is read anyways, our modification only has minimal hardware cost.

Our enhancement improves performance also by reducing the number of instructions and the number of pipeline stalls as well. The instructions for the additional computation unit perform both SRAM reads and an arithmetic operation with the same throughput as the standard RISC instructions. The operations of one `S-ALU` instruction take two or three processor instructions. Therefore, if computations can be placed in `S-ALU`, the number of instructions can be reduced. At the same time, `S-ALU` is placed after the SRAM reads in the pipeline, which eliminates the pipeline stall between the SRAM read and an arithmetic operation.

Obviously, there have been several proposals in the literature and efforts in industrial products to add additional logical functionality to enhance processor performance in embedded and conventional systems. However, previous proposals either require significant hardware modification or are limited to specific applications. Our enhancement only requires small hardware modifications; essentially, just one ALU next to the SRAM with communication paths, and targets general computations rather than stream computation where other SIMD or DSP extensions work extremely well. (For more details, refer to Section 6 for related work).

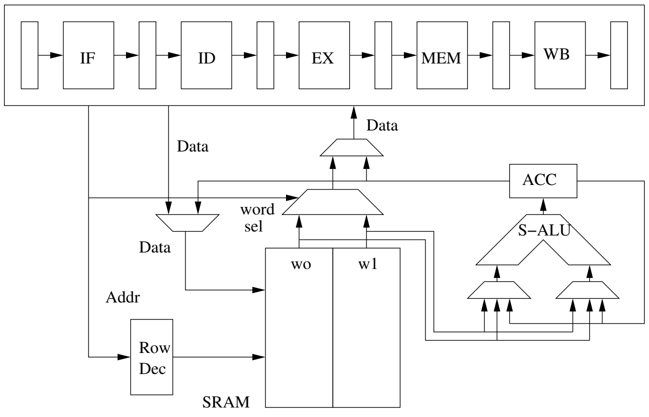

Figure 1: Hardware Architecture

We develop a compiler algorithm that takes advantage of this additional functionality to improve program performance. We use the data flow graph of a program as input to our computation partitioning algorithm that assigns the computations to the processor or to the new computation unit. This algorithm is used before instruction selection, scheduling and register allocation. We describe a branch-and-bound algorithm which uses a cost function that estimates data transfer instruction costs in terms of the number of instructions for any given partitioning. The algorithm generates the SRAM row constraints for data layout that then need to be satisfied by a data layout algorithm. It ensures that the set of generated constraints is feasible.

There are many applications that can benefit from our approach. The performance improvement comes from the reduced memory accesses and the reduction in the number of cycles to perform an operation on SRAM words using the computation unit as compared to using the processor ALU. Our experimental results show that the benefits of this modification is significant although the additional hardware cost is minimal.

The rest of the paper is organized as follows. In Section 2 we describe our approach for the proposed enhancement. In Section 3 we formulate the partitioning problem associated with the proposed enhancement and give a detailed algorithm to solve the partitioning problem. We illustrate our approach using an example in Section 4. In Section 5 we show experimental results on some benchmarks. In Section 6 we provide a summary of the related work. We conclude in Section 7.

2. OVERALL APPROACH

Our goal is to improve embedded system performance by adding a small amount of hardware that augments the functionality of the main processor. We describe the software and hardware aspects of our approach in this section.

2.1 Hardware Architecture

The hardware modification comprises of a small computation unit along with an accumulator near the on-chip SRAM. The hardware architecture of our enhancement is shown in Figure 1. We show a 5-stage pipeline of the processor similar to the one used in the ARM9TDMI [1] processor. The processor pipeline does not require any modifications except for the support needed for the new instructions to use the computation unit. The computation unit operates under the control of the main processor with the help of the new additional instructions.

Instruction	Description
OPS2 Ws1, Ws2, (Imm)Rs	OP(SRAM[<Rs>+Imm,Ws1], SRAM[<Rs>+Imm,Ws2]) → ACC
OPSA (Imm)Rs	OP(SRAM[<Rs>+Imm], ACC) → ACC
OPSA' (Imm)Rs	OP(ACC,SRAM[<Rs>+Imm]) → ACC
MACCR Rd	ACC → REG[Rd]
MRACC Rs	REG[Rs] → ACC
STACC (Imm)Rd	ACC → SRAM[<Rd>+Imm]
LDACC (Imm)Rs	SRAM[<Rs>+Imm] → ACC

Table 1: New Instructions

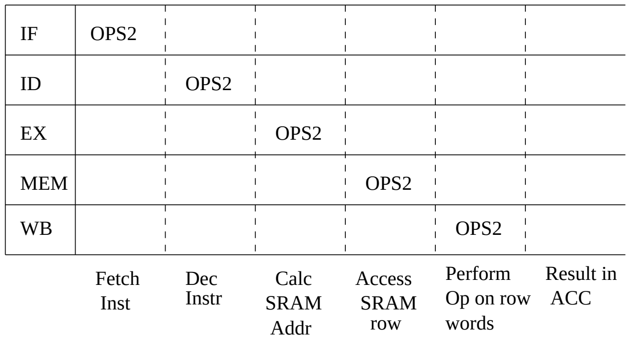

Figure 2: Pipeline Flow

2.2 New Instructions

The new instructions that support the new computation unit are shown in Table 1. In this subsection, we assume that each SRAM row consists of two words. The description can be easily extended to more words per row. The instructions are symmetric in terms of their input operands, i.e., the SRAM word `w0`, `w1`, and the ACC can appear on either input of the `S-ALU`. The `OPS2` instruction takes two words in the same SRAM row, performs an operation `OP`, and stores the result in the accumulator `ACC`. A register (`Rs`) and an immediate (`Imm`) specify a row in the SRAM, and `Ws1` and `Ws2` specify a word within the row. The instruction `OPSA` takes one operand word from the SRAM (`w0` or `w1`) and the other operand from the `ACC`, performs the operation `OP`, and stores the result in the accumulator `ACC`. The instructions `MACCR` and `MRACC` perform data transfer between a processor register and the accumulator. The instructions `STACC` and `LDACC` perform the data transfer between the accumulator and the SRAM.

The new instructions can be compactly encoded with a few opcodes. For example, the encoding of `OPS2` consists of the opcode (6 bits), the address register `Rs` (5 bits), the immediate `Imm` (14 bits), the bits specifying words `Ws1`, `Ws2` (2 bits), and the operation field `OP` (5 bits). Using an operation field, all operations (ADD, SUB, SHIFT, etc.) are encoded with one opcode. Similarly, all other instructions can be encoded using only a few opcodes.

2.3 Pipeline Flow

Figure 2 shows the pipeline flow of the `OPS2` instruction. The instruction is fetched in the IF stage, decoded in the ID stage, the SRAM address is calculated in the EX stage, and the SRAM is accessed in the MEM stage of the pipeline. The above steps are the same as for a processor load instruction. The row data read from the SRAM is fed to `S-ALU`.

The `S-ALU` computation unit performs the operation specified in the `OPS2` instruction during the writeback stage of the pipeline and writes the result into the accumulator. The result of the accumulator is available in the next cycle. Assuming the SRAM latency is 1 cycle, the processor pipeline does not stall for the `OPS2` instruction. The instruction `OPSA` flows through the pipeline in the same manner.

If a standard RISC processor were to perform an operation on two words from the SRAM, it would require two loads from the SRAM and one operation instruction. If a load instruction is followed by an ALU instruction that uses the load value, then the processor would stall for one cycle (assuming 1 cycle SRAM latency) because the value of the load instruction is available at the end of MEM pipeline stage, but the ALU instruction is in the EX stage when the load is in MEM stage. This pipeline stall can be avoided if some other instruction independent of the load value is inserted between the load and the ALU instruction. The instruction `OPS2` saves one load instruction to the processor and one instruction for the operation. The `OPS2` instructions saves two cycles compared to the sequence of instructions needed to perform the same operation on the processor for the two SRAM words, assuming 1 cycle SRAM latency. One saving comes from reading the two words from the same SRAM row access. The other saving comes from performing an operation during the WB stage of the pipeline.

2.4 Software Support

In addition to the conventional compilation steps, we have to perform a partitioning step for any given program that determines which arithmetic unit any given operation will be executed on. Therefore, for a given program, after obtaining the Intermediate Form (IF) representation of the program, we perform the following steps:

1. Generate directed acyclic graphs (DAGs) corresponding to the basic blocks of the program.
2. For each DAG, run the computation partitioning algorithm (cf. Section 3). The algorithm assigns the operation nodes to the processor ALU called `P-ALU` or `S-ALU` and generates the SRAM row constraints.
3. We perform the data layout using a simple algorithm that satisfies the SRAM row constraints.
4. The steps of instruction selection, scheduling and register allocation are performed to generate code.

3. ALGORITHM

We formulate our computation partitioning problem as a cost minimization problem, and describe an algorithm to solve the problem. In the algorithm, it is assumed that there are two ALUs: the ALU near the SRAM (`S-ALU`) and the ALU in the processor (`P-ALU`).

3.1 Partitioning Problem

Given a directed acyclic graph G with the nodes either representing the computation or variables in the SRAM, the partitioning problem is to *assign the computation nodes to* `S-ALU` *or* `P-ALU` *in a way that minimizes the total number of cycles required to execute the program with a feasible data layout.*

To approximate the cycles to execute a program, we define the cost of a partition C as $P_c + S_c + E_c$. P_c and S_c are the number of computation instructions assigned to `P-ALU` and `S-ALU` respectively, which represent the cycles for computations. E_c is the number of instructions required for data transfers to satisfy the data dependencies.

```
/* Initialization */
BestCost = $\infty$
BestSol = (-,-,-)
$C$ = $num\_computation\_nodes(G)$
call $partition\_nodes(G$, -, -, -, $C)$

$partition\_nodes(G, P, S, R, C)$ {
  if (!$all\_assigned(G)$) {
    for each $v$ in $eligible\_nodes(G)$ {
      /* Assign $v$ to P-ALU */
      $P' = P \cup \{v\}$; $G' = assign\_to\_P(G,v)$
      $C' = C + update\_cost(G,G',v)$
      if ($C'$ < BestCost) $partition\_nodes(G',P',S,R,C')$

      /* Assign $v$ to S-ALU (OPSA) */
      $S' = S \cup \{v\}$; $G' = assign\_to\_SA(G,v)$
      $C' = C + update\_cost(G,G',v)$
      if ($C'$ < BestCost) $partition\_nodes(G',P,S',R,C')$

      /* Assigned $v$ to S-ALU (OPS2) */
      $S' = S \cup \{v\}$; $G' = assign\_to\_S2(G,v)$
      $RC = \{\text{row}(inp_1(v)) = \text{row}(inp_2(v))\}$
      succ = $check\_constraint(R, RC)$
      if (succ) {
        $R' = R \cup RC$
        $C' = C + update\_cost(G,G',v)$
        if ($C'$ < BestCost) $partition\_nodes(G',P,S',R',C')$
      }}
  } else {
    if ($C$ < BestCost) {
      BestCost = $C$
      BestSol = $(P, S, R)$}}
}
```

Figure 3: Computation Partitioning Algorithm

```
$update\_cost(G, G', v)$ {
  $change = 0$; $iv_1 = inp_1(v)$; $iv_2 = inp_2(v)$
  $change$ += $data\_transfer\_cost(G', iv_1)$
  $change$ -= $data\_transfer\_cost(G, iv_1)$
  $change$ += $data\_transfer\_cost(G', iv_2)$
  $change$ -= $data\_transfer\_cost(G, iv_2)$
  return $change$
}
```

Figure 4: Lower Bound Cost Computation

Since we assume that the cost C represents the cycles taken for an execution, the goal of our partitioning algorithm is to find the partition between `P-ALU` and `S-ALU` that minimizes C. We use a branch-and-bound algorithm to achieve this goal.

3.2 Computation Partitioning Algorithm

The algorithm takes a DAG with computation and data variable nodes as input and generates as output the assignment of the computation nodes to `P-ALU` and `S-ALU` and the necessary SRAM row data layout constraints.

The algorithm is shown in Figure 3. In the algorithm, G is the data flow graph, P is the set of nodes assigned to `P-ALU`, S is set of nodes assigned to `S-ALU`, C is cost of the current assignment, and R is the set of SRAM row constraints. In this algorithm, a node is eligible for assignment only when all its inputs have been assigned. The branch-and-bound algorithm uses a lower bound on the cost of current partial assignment to prune the search space.

The leaf nodes represent the variables residing in the SRAM and these nodes are considered as the assigned nodes in the graph G. The best cost is initialized to ∞. The assignment of nodes and the row constraints is initialized to an empty set, which represents the best solution. The initial cost C is the number of computation nodes in G, which represents $P_c + S_c$.

If there is a node yet to be assigned, then the algorithm tries to assign each eligible node to `P-ALU`, `S-ALU` with `OPS2`, and `S-ALU` with `OPSA` one at a time. To try a new assignment, the node is added to the set P' (or S') and marked as assigned in G'. The lower bound of the current partial assignment is computed (C'). The partitioning algorithm is called recursively only if the current cost is lower than the current best solution. For the assignment to `S-ALU` with `OPS2`, the algorithm first checks if both input nodes can be placed in the same SRAM row with the constraints already given. The assignment is tried only when the row constraint can be satisfied.

If all the nodes are assigned, then the current assignment represents one possible solution and the current cost (C) represents the cost of the solution. If the cost is less than that of the current best solution, the best cost and the best solution is updated with new values.

3.3 Cost Computation

The cost of a partition consists of three components: computation cost in `P-ALU` (P_c), computation cost in `S-ALU` (S_c), and data transfer cost (E_c). The computation cost ($P_c + S_c$) is simply the number of computation nodes in the graph G.

The data transfer cost due to data flow dependencies can be computed from the graph G and the assignment of its nodes. For example, consider the graph in Figure 5 (a). If the addition node that uses A and B is assigned to `P-ALU`, there is the data transfer cost of two instructions because both A and B should be moved from the SRAM to the register file. Similarly, if the second addition node E is assigned to `S-ALU`, there is the cost of one instruction because the result of the first addition should be moved to either the accumulator or the SRAM.

For each node v, the algorithm finds the data transfer cost of the node by looking at the assignment of the following nodes that use the result from v (we refer to these as *fanout nodes* of v). Table 2 summarizes the cost for each case. For a node under consideration, let n_p the number of fanout nodes assigned to the processor, n_{sa} be the fanout nodes assigned to the `S-ALU` with `OPSA`, and n_{s2} be the number of fanout nodes assigned to the `S-ALU` with `OPS2` operation. Essentially, whenever the result of an operation at `P-ALU` is used by `S-ALU` or vice versa, one data transfer is required. Also, when the value in the accumulator is used by the `OPS2` instruction, it should be written to the SRAM.

With a partially assigned graph, the computed cost should be the lower bound that holds under any assignment of the unassigned nodes so that it can be used to prune bad partitions. Therefore, the data transfer cost of unassigned nodes is assumed to be zero. Because a node is eligible for assignment only when all its inputs have been assigned, none of the fanout nodes of an unassigned node can be assigned. In the best case, the algorithm may be able to assign all remaining nodes so that the data transfer cost of those nodes is zero.

v	Condition	(Data Transfer)
`P-ALU`	$n_{sx} + n_{s2} \geq 1$	+1
	otherwise	0
`S-ALU`	$n_p \geq 1$ and $n_{s2} \geq 1$	+2
	$n_p \geq 1$ xor $n_{s2} \geq 1$	+1
	otherwise	0
SRAM	$n_p \geq 1$	+1
	otherwise	0

Table 2: Templates for data transfer costs.

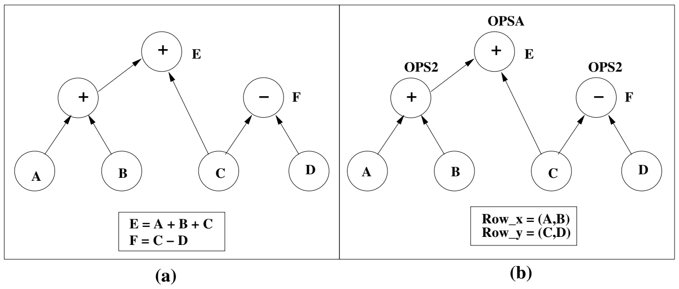

Figure 5: Basic Block Example

In our algorithm, the cost is computed incrementally as we assign each node as shown in Figure 4. When a new node is assigned to either `P-ALU` or `S-ALU`, it can change the data transfer cost of its two input nodes. Therefore, the algorithm computes the change in the data transfer cost of the input nodes for each new node assigned, and adds the change to the current cost to obtain the new cost.

3.4 SRAM Data Layout

The SRAM row constraints generated by the computation partitioning algorithm are satisfied by performing data layout for the variables that would reside in SRAM. The data layout algorithm takes the row constraints and assigns the variables that are part of a constraint to the same SRAM row. In the case where a group of variables are part of an array, then the address assignment for the array is done such that the SRAM row constraints are satisfied.

4. EXAMPLE

Figure 5(a) shows the C code for an example basic block and the data flow graph for the basic block. The assignment of the computation nodes to the processor and the computation unit based on the computation partitioning algorithm is shown in Figure 5(b) along with the SRAM row constraints generated by the algorithm. The instruction schedule with only the processor instructions has a total of 8 instructions: 4 loads, 2 ADD operations, 1 SUB operation, and 2 stores. The Instruction schedule with both `P-ALU` and `S-ALU` has a total of 5 instructions: 1 `(ADDS2)`, `(SUBS2)`, 1 `(ADDSA)`, and 2 `STACC` instructions. Assuming the SRAM latency of 1 cycle, the instruction schedule using only the base processor instructions takes 16 cycles whereas it takes 12 cycles for the instruction schedule of the partitioned graph when both ALUs are used. In this example, the use of both `P-ALU` and `S-ALU` save 4 cycles giving a 33% speedup.

5. EXPERIMENTAL RESULTS

We provide experimental results on a collection of programs. We assume a 5-stage processor pipeline similar to the one in ARM9TDMI [1]. We assume that the computation unit `S-ALU` can perform all the arithmetic and logical operations as in the `P-ALU` except that it does not have a multiplier function and that the processor multiply instruction takes 4 cycles. The hardware cost of the `S-ALU` would be approximately 3K gates.

Some information about the benchmarks is shown in Table 3. The programs `plus` and `leaf_comp` are the modules from the MPEG decoder application. In the `plus` module, a difference block is added to the reference block to generate the output image. In the `leaf_comp` module, a sequence of operations is performed on the leaf nodes stored in the memory. The `AVG` program computes the average value of an input array. The `VADD` program adds the corresponding elements of two input arrays and writes the result into a new output array. `MatrixAdd` is a two-dimensional matrix addition program. `SHA1` is the secure hash algorithm to compute a hash value of the input data. `MD5` is a message digest algorithm that computes a message digest of the input. `GSM_DEC` is a code segment from a GSM decoder module.

We first generate the DAGs for the basic blocks of the benchmarks. The total number of nodes (DagN) in the DAGs of the programs and the number of operation nodes (OpN) in the DAGs of the programs are shown in Table 3. The DAGs are given as input to our branch-and-bound computation partitioning algorithm. The algorithm takes 27ms to 200ms to partition the benchmarks on a 1GHz Pentium machine.

The assignment of the DAG nodes is shown in Table 3. The columns `OPS2` and `OPSA` show the number of nodes assigned to the SRAM computation unit (`S-ALU`) using the corresponding instruction. The column `PrOP` shows the number of nodes assigned to be performed on the processor (`P-ALU`). Note that SRAM row constraints are generated for the benchmarks `plus`, `VADD`, and `MatrixAdd` because of an operation node assignment to use the `OPS2` instruction.

We obtain an instruction schedule for the original case (without the SRAM computation unit) and the new case (with the SRAM computation unit). In the original case, all the DAG operation nodes are assigned to the processor. In the new case, the DAG operation nodes are assigned using the computation partitioning algorithm as shown in Table 3. Then, we compute the number of cycles for these instruction schedules using the number of cycles per instruction including the pipeline stall cycles and the number of cycles for the data transfer instructions. We show the performance of the benchmarks for the original case (Orig) and the new case (New) for different SRAM latencies (1, 2, and 3 cycles) in Table 4. The columns (Speedup) show the speedup in performance for a given SRAM latency. The speedup is obtained by dividing the number of cycles for the original case by the number of cycles for the new case. The speedup ranges from 1 to 1.48 for the benchmarks.

The performance speedup for the benchmark varies with the SRAM latency. For the benchmarks `plus`, `VADD`, and `MatrixAdd`, the speedup increases for higher SRAM latency because these benchmarks have an operation node assigned to use the `OPS2` instruction and the benefit of reducing an SRAM load operation in the `OPS2` instruction increases for higher SRAM latency. But for the benchmarks `AVG`, `SHA1`, `MD5`, and `GSM_DEC`, the speedup decreases for the higher SRAM latency because the computation in these benchmarks results in using the new computation unit with the `OPSA` instruction only. So, the relative benefit of the reduced number of instructions and the pipeline stalls offered by the `OPSA` instruction decreases with the increase in the SRAM latency.

Benchmark	DagN	OpN	`OPS2`	`OPSA`	`PrOP`
`plus`	8	4	1	0	3
`VADD`	6	2	1	0	1
`AVG`	6	2	0	1	1
`leaf_comp`	7	3	1	1	1
`MatrixAdd`	13	5	1	0	4
`SHA1`	36	20	0	3	17
`MD5`	19	11	0	1	10
`GSM_DEC`	15	9	0	9	0

Table 3: Benchmarks and Computation Node Assignments

The speedup for the benchmarks depends on the number of executed `OPS2` and `OPSA` instructions as a fraction of the total number of executed instructions. For example, in `SHA1`, the first part of the benchmark benefits from the new computation unit, but the second part gets assigned to the processor. So, even though the first part has a speedup of 1.26, the overall speedup is 1.09 (for SRAM latency = 1 cycle). Also, the benefit of the `OPS2` and `OPSA` instructions may be offset by the data transfer overhead cycles. For example, in `plus`, the benefit of using the `OPS2` instruction is offset by the data transfer overhead cycles for the 1 cycle SRAM latency (speedup 1.0). But, for the higher SRAM latency the the speedup improves because the benefit of using the `OPS2` instruction is more than the data transfer overhead cycles.

6. RELATED WORK

Some embedded RISC processors have been extended with new instructions to support digital signal processing and/or multimedia processing functions (e.g., ARM DSP and SIMD extensions [6, 3], Mediabreeze [14], and DSP-RAM [15]). In order to use the data level parallelism in multimedia applications, some microprocessors have been extended with vector processing instructions [4, 8]. In another approach, a reconfigurable coprocessor (e.g., Piperench [5]) or a configurable and extensible processor (e.g., Xtensa [2]) is used to improve application performance. The Xtensa processor allows the base processor to be tailored to match the application requirements by selecting and configuring predefined elements of the architecture or extending the processor with additional execution units and the required support. A problem with the above approaches is that these extensions have high hardware cost and require significant changes in some cases to the RISC processor microarchitecture. Our approach requires much less hardware and changes to the processor pipeline to support the SRAM computation unit.

In order to bridge the gap between the processor speed and memory latency, there are approaches that incorporate some level of processing capability near the off-chip DRAM (e.g., Active Pages [11], IRAM [7]). These approaches are not suitable for embedded systems due to the hardware cost overhead. We put a small computation unit near the *on-chip* SRAM to reduce some of the memory data transfers between the RISC processor and the on-chip SRAM.

Benchmark	SRAM latency = 1 cycle			SRAM latency = 2 cycles			SRAM latency = 3 cycles		
	Orig (cycles)	New (cycles)	Speedup	Orig (cycles)	New (cycles)	Speedup	Orig (cycles)	New (cycles)	Speedup
`plus`	1438	1438	1.00	1738	1638	1.06	2040	1840	1.10
`VADD`	803	603	1.33	1105	805	1.37	1407	1007	1.39
`AVG`	609	409	1.48	710	510	1.39	811	611	1.32
`leaf_comp`	904	704	1.28	1207	907	1.33	1510	1110	1.36
`MatrixAdd`	2437	1925	1.26	3208	2440	1.31	3979	2955	1.34
`SHA1`	3002	2746	1.09	3339	3083	1.08	3676	3418	1.07
`MD5`	896	832	1.07	1024	960	1.06	1152	1088	1.05
`GSM_DEC`	800	600	1.33	1100	900	1.22	1400	1200	1.16

Table 4: Comparison of Benchmark Performance

In the digital signal processors (DSPs), memory is organized as two memory banks. The problem of partitioning the variables between the two memory banks and register allocation is studied in [9, 13, 12]. In the DSPs with SIMD memory accesses, the data are organized in groups whose elements share a common address and an approach for memory layout of variables is presented in [10]. In our approach, we solve the computation node partitioning problem and the SRAM row constraints representing the data layout constraints are a result of our algorithm.

7. CONCLUSIONS

We enhanced embedded system functionality by adding a small computation unit with an accumulator near the on-chip SRAM. The computation unit requires a few additional instructions and operates under the control of the processor pipeline. We provided an algorithm that allows the computation to be partitioned between the processor and the computation unit. Our experimental results show that our small modification to the embedded system can provide significant performance improvement. Our approach offers a good balance between the hardware cost, hardware and software modifications, and performance improvement. Our proposed ISRAM enhancement can also reduce energy consumption in the embedded system by reducing the accesses to the SRAM and the register file.

There are some limitations of our ISRAM approach. Since the computation partitioning and the SRAM data layout for data subject to row constraints is done at compile time, our approach is currently only applicable to statically allocated data structures. In the partitioning algorithm, the computation nodes assigned to the computation unit may be limited due to data layout constraints that need to be satisfied. The benefits of using the SRAM computation unit enhancement depend on a given program's computation because some of the functions (e.g., multiply) may not be available on the computation unit to keep the hardware overhead low.

Our approach can be extended in several ways. For example, the SRAM computation unit can be extended with some control logic to support counters and address generation for multiple operations using the accumulator and the SRAM. If the SRAM is organized as multiple memory banks, the computation unit can be extended to operate on two words from different memory banks. In embedded systems with caches, our approach can be extended to caches where the operations would be performed on two words of a cache line or two words from different cache lines in the same set.

8. REFERENCES

[1] The ARM9TDMI Technical Reference Manual Rev 3. *ARM Limited http://www.arm.com*, 2000.

[2] Xtensa Architecture and Performance. *Tensilica, Inc. http://www.tensilica.com*, September 2002.

[3] David Brash. The ARM Architecture Version 6. *ARM Limited http://www.arm.com*, January 2002.

[4] K. Diefendorff, P. K. Dubey, R. Hochsprung, and H. Scale. AltiVec Extension to PowerPC Accelerates Media Processing. *IEEE Micro*, 20(2):85–95, 2000.

[5] S. Goldstein *et. al.* Piperench: A coprocessor for streaming multimedia acceleration. In *26th Int'l Symposium on Computer Architecture*, 1999.

[6] Hedley Francis. ARM DSP-Enhanced Extensions. *ARM Limited http://www.arm.com*, May 2001.

[7] Chistoforos E. Kozyrakis, Stylianos Perissakis, and David Patterson. Scalable Processors in the Billion-Transistor Era: IRAM. *IEEE Computer*, 30(9):75–78, September 1997.

[8] Corinna G. Lee and Mark G. Stoodley. Simple Vector Microprocessors for Multimedia Applications. In *Proceedings of the 31st Annual ACM/IEEE International Symposium on Microarchitecture*, 1998.

[9] Rainer Leupers and Daniel Kotte. Variable partitioning for dual memory bank dsps. In *IEEE International Conference on Acoustics, Speech, and Signal Processing*, volume 2, pages 1121–1124, 2001.

[10] Markus Lorenz, David Kottmann, Steven Bashford, Rainer Leupers, and Peter Marwedel. Optimized Address Assignment for DSPs with SIMD Memory Accesses. In *Proceedings of the ASP-DAC*, 2001.

[11] Mark Oskin, Frederic T. Chong, and Timothy Sherwood. Active pages: A computation model for intelligent memory. In *25th International Symposium on Computer Architecture*, pages 192–203, 1998.

[12] Amit Rao and Santosh Pande. Storage Assignment Optimizations to Generate Compact and Efficient Code on Embedded DSPs. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*, pages 128–138, May 1999.

[13] Ashok Sudarsanam and Sharad Malik. Memory Bank and Register Allocation in Software Synthesis for ASIPs. In *IEEE/ACM International Conference on Computer-Aided Design*, pages 388–392, 1995.

[14] Deependra Talla and Lizy K. John. Cost-effective Hardware Acceleration of Multimedia Applications. In *Proceedings of the IEEE International Conference on Computer Design*, 2001.

[15] Zixiong Wang, Bruce F. Cockburn, Duncan G. Elliott, and Witold A. Krzymein. DSP-RAM: A Logic-Enhanced Memory Architecture for Communication Signal Processing. In *IEEE Pacific Rim Conference on Communications, Computers and Signal Processing*, pages 475–478, 1999.