UC Irvine
ICS Technical Reports

Title
Memory subsystem description in EXPRESSION

Permalink
https://escholarship.org/uc/item/5qh4b1zz

Authors
Mishra, Prabhat
Grun, Peter
Dutt, Nikil
et al.

Publication Date
2000-10-05

Peer reviewed

eScholarship.org Powered by the California Digital Library
University of California

Notice: This Material
may be protected
by Copyright Law
(Title 17 U.S.C.)

ICS

TECHNICAL REPORT

Memory Subsystem Description in EXPRESSION

Prabhat Mishra, Peter Grun, Nikil Dutt, and Alex Nicolau
{pmishra, pgrun, dutt, nicolau}@ics.uci.edu
http://www.cecs.uci.edu/~aces

UCI-ICS Technical Report #00-31
Dept. of Information and Computer Science
University of California, Irvine, CA 92697

October 5, 2000

Information and Computer Science
University of California, Irvine

Memory Subsystem Description in EXPRESSION

Prabhat Mishra
pmishra@ics.uci.edu

Peter Grun
pgrun@ics.uci.edu

Nikil Dutt
dutt@ics.uci.edu

Alex Nicolau
nicolau@ics.uci.edu

Architectures and Compilers for Embedded Systems (ACES) Laboratory
Center for Embedded Computer Systems
University of California, Irvine, CA, USA
http://www.cecs.uci.edu/~aces

Technical Report #00-31
Dept. of Information and Computer Science
University of California, Irvine, CA 92697, USA

October 2000

Abstract

Memory represents a major bottleneck in modern embedded systems. Traditionally, memory organizations for programmable systems assumed a fixed cache hierarchy. With the widening processor-memory gap, more aggressive memory technologies and organizations have appeared, allowing customization of a heterogeneous memory architecture tuned for the application. However, such a processor-memory co-exploration approach critically needs the ability to explicitly capture heterogeneous memory architectures. We present in this report the mechanism for describing memory subsystems in EXPRESSION, an Architecture Description Language(ADL) for processor-memory systems. The memory subsystem for the retargetable simulator can be generated from the description automatically. We have demonstrated the technique by generating memory subsystems for C6x, R10K, Itanium and PowerPC architectures. We present a set of experiments using our memory aware ADL Language to drive the exploration of the memory subsystem for the TIC6211 processor architecture, demonstrating a range of cost and performance attributes.

Contents

1 Introduction 3

2 Our Approach 3

3 Survey of Contemporary Memory Architectures 4

4 EXPRESSION description of the memory subsystem 6

5 C6x Memory Subsystem Description 8

6 Memory Subsystem Simulator Generation 12

7 Experiments 14
- 7.1 Experimental Setup 15
- 7.2 Cost Estimation of the memory configurations 15
- 7.3 Results 17

8 Conclusion 19

9 Acknowledgements 22

List of Figures

1 The Flow in our approach 4
2 Sample Memory Architecture for the TIC6211 10
3 Memory Subsystem 12
4 Cycle counts for the memory configurations on TIC6211 17
5 Cycle accounting for memory configuration 1 18
6 Cycle accounting for memory configuration 2 19
7 Cycle accounting for memory configuration 3 20
8 Cycle accounting for memory configuration 4 20
9 Cycle accounting for memory configuration 5 21
10 Cycle accounting for memory configuration 6 21

List of Tables

1 Memory features of different architectures. *c6x: TI C6x, MAP: MAP1000A, SC: Starcore, AV: Motorola Altivec, U3: SUN UltraSparc III, PA: PA 8500, Alpine: QED Alpine RM57x, v830: NEC V830, SA: StrongArm, α364: Alpha 364* 6
2 Memory subsystem primitives 7
3 The memory subsystem configurations 16

1 Introduction

The advent of System-on-Chip (SOC) technology has resulted in a paradigm shift for the design process of embedded systems employing programmable processors with custom hardware. Modern system-level design libraries frequently consist of Intellectual Property (IP) blocks such as processor cores that span a spectrum of architectural styles, ranging from traditional DSPs and superscalar RISC, to VLIW and hybrid ASIPs. Furthermore, SOC technologies permit the incorporation of novel on-chip memory organizations (including the use of on-chip DRAM, frame buffers, streaming buffers and partitioned register files), allowing a wide range of memory organizations and hierarchies to be explored and customized for the specific embedded application.

Recent work on language-driven Design Space Exploration (DSE) ([1], [3], [4], [5], [6], [10], [27], [28], [30]), uses Architectural Description Languages (ADL) to capture the processor architecture, generate automatically a software toolkit (including compiler, simulator, assembler) for that processor, and provide feedback to the designer on the quality of the architecture. While these approaches extensively address processor features (such as instruction set, number of functional units, etc.) to our knowledge no previous approach allows explicit capture of a customized, heterogeneous memory architecture, and the attendant tasks of generating a software toolkit that fully exploits this memory architecture. In this report we show how to describe memory subsystem in EXPRESSION [10] ADL. A Memory aware compiler and simulator is generated automatically from the EXPRESSION description of processor and memory subsystem.

Section 2 outlines our approach and the overall flow of our environment. Section 3 surveys the contemporary memory architectures. Section 4 presents the memory access abstractions necessary to describe variety of memory configurations and describes the primitives used in EXPRESSION ADL to describe the memory subsystem abstractions. Section 5 presents how to describe the C6x memory subsystem using these primitives. Section 6 describes implementation details. Section 7 illustrates memory architecture exploration using experiments on the TIC6211 processor, with varying memory configurations to trade-off cost versus performance. Section 8 concludes the report.

2 Our Approach

Figure 1 shows the flow in our approach. In our IP library based Design Space Exploration (DSE) scenario, the designer starts by selecting a set of components from a processor IP library and memory IP library. Our EXPRESSION Architectural Description Language (ADL) description (containing a mix of such IP components and custom blocks) is then used to generate the information necessary to target both the compiler and the simulator to the specific processor-memory system.

Traditionally, the memory subsystem was transparent [1] to the processor and the software toolkit. While the processor pipeline was captured in detail to allow aggressive scheduling in the compiler, the memory subsystem pipeline was not explicitly captured and exploited by the compiler. How-

[1] i.e., assumed an implicitly defined memory architecture, e.g., a fixed cache hierarchy

ever, by describing the pipelining and parallelism available in recent memory organizations, there is tremendous opportunity for the compiler to generate performance and power improvements. Indeed, as shown in Figure 1, our previous work on RTGEN [9] (Reservation Tables generation algorithm) and TIMGEN [7] (Timing Generation algorithm) already generates the timing information for both the processor and memory subsystem pipelines starting from the ADL description of the memory. The compiler uses this detailed timing information to hide the latency of the lengthy memory operations in the presence of efficient memory access modes (e.g., page/burst modes), and cache hierarchies [8], to generate significant performance improvements. Such aggressive optimizations are only possible due to the explicit representation of the detailed memory architecture.

We present here the memory subsystem description in EXPRESSION, along with the abstractions that allow capturing a set of heterogeneous memory modules, and connecting them to form customized memory architectures. Furthermore, in a DSE environment it is crucial to provide the designer with detailed feedback on the choices made in the processor and memory architectures. In [24] we presented our cycle-accurate structural simulator generation approach for the processor descriptions in EXPRESSION. In this paper we present the memory simulator generation (shown shaded in Figure 1) that is integrated into the SIMPRESS simulator, allowing for detailed feedback on the memory subsystem architecture and its match to the target applications.

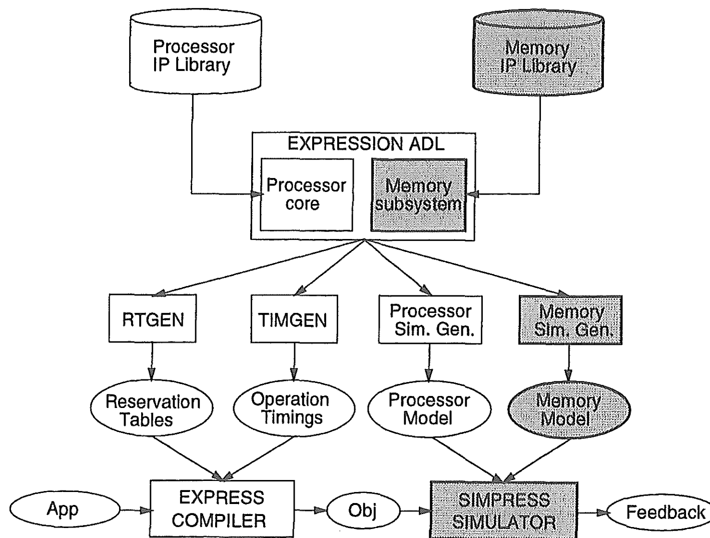

Figure 1. The Flow in our approach

3 Survey of Contemporary Memory Architectures

Modern processors cover a wide spectrum of architectural styles, ranging from traditional DSPs and superscalar RISC, to VLIW and hybrid ASIPs. Each architectural style comes with a wide variety of memory configurations. In the following, we review representative architectures for each class.

Motorola 56K [25] and TI C5x [29] belong to traditional **DSP** area. Motorola 56K has two data paths. 2 24-bit data registers in each path (x0, x1, y0, and y1). Register x is the concatentation of x1 and x0 and can store 48 bits of an accumulator. Likewise for register y. DSP56002 [13] permits simultaneous accesses to on-chip program and two data memories (Harvard architecture). It has 512 x 24-bit on-chip Program RAM, 64 x 24-bit bootstrap ROM, two 256 x 24-bit on-chip data RAMs, two 256 x 24-bit on-chip data ROMs containing sine, A-law and μ-law tables. External memory expansion is done using 16-bit address and 24-bit data buses. Bootstrap loading is possible from external data bus, Host Interface, or Serial Communications Interface. TI C5x [29] has on-chip 16 x 1056 dual access RAM, variable size on-chip single access RAM (0 to 16 x 9K), and variable size on-chip ROM (16x2K to 16x32K) with bootloading facility in most of the configurations.

TI C6x [22], MAP1000A [2] and Starcore [16] lie in **VLIW DSP** area. TIC6x [22] family comes with on-chip configurable SRAM and off-chip DRAM with page/burst access modes. They support DMA and parallel data transfers. They differ in terms of cache support. C6201 [22] and C6202 [22] does not have any data cache whereas C6211 [22] and C6711 [22] have 2 levels of data cache. MAP100A has data cache and 4K SRAM. It supports a data streamer which enables on chip memory modules to route data from the main memory to the CPU. It has 64 address generation channels and 64 FIFO pointers. In Starcore, memory is external to the SC140 core, and can be configured in many ways, e.g., 32K groups on-chip unified data/program memory and no data cache. Memory accessed through 2 data buses (XDBA, AXDBB) and one program bus(PDB).

Trimaran [30], IA-64 [11] and PA 8500 [15] belong to the **EPIC** category. They come with prefetch capabilities to bring the data earlier into the cache, to insure a hit. They have multiple levels of caches. IA-64 supports data speculation which moves loads earlier, possibly past store instructions, breaking dependencies. ALAT table stores 32 such simultaneous loads which need to be checked for validness.

Superscalar processors like Motorola Altivec [18] and SUN UltraSparc III [21] allow cache prefetches. In UltraSparc, on-chip memories allow a software-controlled cache behavior. They support multiple level of data caches. Alpha 364 has victim buffer (sometimes called cache) for better cache performance. PowerPC [17] and R10000 [20] processors belongs to this category.

RISC processors like StrongARM [12], NEC V830 [14] and QED Alpine RM57x [19] have on chip SRAM and off-chip DRAM. QED Alpine RM57x and NEC V380 have write cache buffer and DMA support. StrongARM has prefetch buffer and read buffers. The four 32 byte read buffers are loaded explicitly by software.

Table 1 summarizes the memory features for different architectures. Each row of the table corresponds to particular memory feature. Each column shows how many of the memory features are supported by a particular processor. An entry in this table, TAB[F, A], represents the behavior of an architecture *A* towards an memory feature *F*. If an entry is marked *x* then that feature is supported by that architecture. If an entry is blank then that feature is not supported by that architecture. An entry containing and integer number *n* means that features is supported *n* times. For example, the table entry with memory feature *Levels of Data Cache* (second row) and processor name *IA64*

(column 14) has value 3, This implies IA-64 has 3 levels of data cache. The *?* mark in an entry means that it is not obvious whether it supports or not. Motorola 56K and TI C5x are not shown in the table since they do not have any of the memory features shown in the table.

Architecture	C6201	C6211 C6711	MAP	SC	AV	U3	PA	Alpine	V830	SA	R10K	IA64	α364
Levels of Data Cache	0	3	1	0	2	2	2	0	1	2	2	3	2
cache prefetch					x	x	x			x		x	?
cache hints					x							x	?
On-chip SRAM	x	x	x	x		x		x	x	x			
configurable SRAM size	x	x											
Off-chip DRAM	x	x	x	x		x	x	x	x	x			
page/burst modes	x	x	?	?					x				
Write Cache Buffer						x	x	x	x				x
Victim Buffer													x
Stack													
FIFO			x										
Frame buffer													
DMA	x	x	x					x	x				
# transfers in parallel	2	2	?	2	?		2	2			1	2	?
pipelining					?		x				x	?	?

Table 1. Memory features of different architectures. *c6x: TI C6x, MAP: MAP1000A, SC: Starcore, AV: Motorola Altivec, U3: SUN UltraSparc III, PA: PA 8500, Alpine: QED Alpine RM57x, v830: NEC V830, SA: StrongArm, α364: Alpha 364*

4 EXPRESSION description of the memory subsystem

We describe the memory subsystem in EXPRESSION using a set of abstractions, representing the different capabilities of the memory modules. Each module in the memory subsystem is characterized by the following two capabilities: storage capability (allowing to store a set of data) and transfer capability (permitting the unit to transfer data between different modules). For instance, caches represent a fast storage, controlled dynamically by the hardware, which contains both a storage capability (the cache data and tag blocks) and a transfer capability (when a tag is not found, it initiates a cache line fill from the next level). Similarly, a stream buffer contains a storage capability (the FIFO queue containing the data) and a transfer capability (when it recognizes a stream pattern in the sequence of memory accesses, it starts prefetching the data from the next level of

the memory hierarchy). Alternatively, a DMA contains only a transfer capability (which transfers the block of data from the source address to the destination address). The storage capability is characterized by the organization of the storage (e.g., word size, line size, number of lines), while the transfer capability is characterized by the timings and resources used.

The connectivity between the memory modules is represented explicitly, as a netlist. The pipelining and parallelism available in the memory subsystem is described using the pipeline paths and data transfer paths in EXPRESSION [9].

Table 2 shows the primitives used in the memory subsystem description. The first column represents the parameter, the second column represents the possible values for that parameter, and the third column describes the meaning of the parameter.

Parameter	Values	Operation
STORAGE_SECTION		Marks the start of memory subsystem description
TYPE	SRAM, DRAM, REGFILE, DCACHE	Type of the component
TYPE (Cont.)	ICACHE, DMA, WRITE_BUFFER,	Type of the component
TYPE (Cont.)	VICTIM_BUFFER, STREAM_BUFFER	Type of the component
TYPE (Cont.)	CONNECTIVITY	Type of the component
SIZE	positive integer	Number of storage locations
WIDTH	positive integer	Num of bits in each storage
ADDRESS_RANGE	two positive int(start and end address)	Range of addresses
WORDSIZE	positive integer	Number of bits in a word
LINESIZE	positive integer	Num of words in a cache line
NUM_LINES	positive integer	Number of lines in a cache
ASSOCIATIVITY	positive integer	Associativity of the cache
REPLACEMENT_POLICY	LRU, FIFO	Replacement policy for the cache
WRITE_POLICY	WRITE_BACK, WRITE_THROUGH	Write policy for the cache
ENDIAN	LITTLE, BIG	Endianness
READ_LATENCY	positive integer	Time for reading
WRITE_LATENCY	positive integer	Time for writing
NUM_BANKS	positive integer	Number of banks in the memory module
ACCESS_MODE	PAGE, BURST, NORMAL	Access modes supported
NUM_PARALLEL_READ	positive integer	Num of parallel reads per cycle
NUM_PARALLEL_WRITE	positive integer	Num of parallel writes per cycle
READ_WRITE_CONFLICT	0,1	1 means Read/Write conflict for same bank at same time
PIPELINE	pipeline stages	describes the pipeline paths
ACCESS_MODES	page, burst, pipelined etc.	memory access modes

Table 2. Memory subsystem primitives

The memory subsystem is described within STORAGE_SECTION of the EXPRESSION description. In the following sample STORAGE_SECTION description first one is for register file (RFA) description, second one is scratch pad SRAM description, third one is for L1 CACHE description and final one is main memory description. Note that RFA, ScratchPad, L1 and MainMemory are names of the components. They are not parameters. Connections between memory modules can be described structurally as a netlist in terms of ports and connections or behaviorally as shown in the example below in terms of list of storage connections.

```
(STORAGE_SECTION
        (RFA
                (TYPE REGFILE)
                (SIZE 32)
                (WIDTH 32)
        )
        (ScratchPad
                (TYPE SRAM)
                (SIZE 16384)
                (WIDTH 8)
                (ADDRESS_RANGE 0 16383)
        )
        (L1
               (TYPE DCACHE)
               (WORDSIZE 64)
               (LINESIZE 32)
               (NUM_LINES 1024)
               (ASSOCIATIVITY 2)
               (REPLACEMENT_POLICY LRU)
               (WRITE_POLICY WRITE_BACK)
               (LEVEL 1)
               (ADDRESS_RANGE 0 32K)
               (NEXT_LEVEL 2)
        )
        (MainMemeory
                (TYPE DRAM)
                (SIZE 4M)
                (WIDTH 8)
                (LEVEL 3)
        )
        (Connect
                (TYPE CONNECTIVITY)
                (STORAGE_CONNECTIONS (L1, MainMemory))
        )
)
```

5 C6x Memory Subsystem Description

We illustrate our Memory-Aware Architectural Description Language (ADL) using the Texas Instruments TIC6211 VLIW DSP processor that has several novel memory features.

Figure 2 shows the example architecture, containing an off-chip DRAM, an on-chip SRAM, and two levels of cache (L1 and L2), attached to the memory controller of the TIC6211 processor [2]. TI C6211 is an 8-way VLIW DSP processor with a deep pipeline, composed of 4 fetch stages (PG, PS, PR, PW), 2 decode stages (DP, DC), followed by the 8 functional units. The D1

[2]For illustration purposes we present only the D1 ld/st functional unit of the TIC6211 processor, and we omitted the External Memory Interface unit from the figure

load/store functional unit pipeline is composed of D1_E1, D1_E2, and the 2 memory controller stages: MemCtrl_E1 and MemCtrl_E2.

The L1 cache is a 2-way set associative cache, with a size of 64 lines, a line size of 4 words, and word size of 4 bytes. The replacement policy is Least Recently Used (LRU), and the write policy is write-back. The cache is composed of a TAG_BLOCK, a DATA_BLOCK, and the cache controller, pipelined in 2 stages (L1_S1, L1_S2). The cache characteristics are described as part of the STORAGE_SECTION in EXPRESSION [10]:

```
(L1_CACHE
  (TYPE DCACHE)
  (NUM_LINES 64)
  (LINESIZE 4)
  (WORDSIZE 4)
  (ASSOCIATIVITY 2)
  (REPLACEMENT_POLICY LRU)
  (WRITE_POLICY WRITE_BACK)
  (SUB_UNITS TAG_BLOCK DATA_BLOCK L1_S1 L1_S2)
)
```

The memory subsystem instruction set description is represented as part of the Operation Section in EXPRESSION [10]:

```
(OPCODE LDW (OPERANDS (SRC1 reg) (SRC2 reg) (DST reg))
```

The internal memory subsystem data transfers are represented explicitly in EXPRESSION as operations. For instance, the L1 cache line fill from L2 triggered on a cache miss is represented through the LDW_L1_MISS operation, with the memory subsystem source and destination operands described explicitly:

```
(OPCODE LDW_L1_MISS (OPERANDS (SRC1 reg) (SRC2 reg) (DST reg)
  (MEM_SRC1 L1_CACHE) (MEM_SRC2 L2_CACHE) (MEM_DST1 L1_CACHE))
```

This explicit representation of the internal memory subsystem data transfers (traditionally not present in ADLs) allows the designer to reason about the memory subsystem configuration. Furthermore it allows the compiler to exploit the organization of the memory subsystem, and the simulator to provide detailed feedback on the internal memory subsystem traffic[3].

The pipelining and parallelism between the cache operations is described in EXPRESSION through PIPELINE_PATHS [10]. Pipeline Paths represent the ordering between pipeline stages in the architecture (represented as bold arrows in Figure 2). For instance, a load operation to a DRAM address traverses first the 4 fetch stages (PG, PS, PR, PW) of the processor, followed by the 2 decode stages (DP, DC), and then it is directed to the load/store unit D1. Here it traverses the D1_E1 and D1_E2 stages, and is directed by the MemCtrl_E1 stage to the L1 cache, where it traverses the L1_S1 stage. If the access is a hit, it is then directed to the L1_S2 stage, and the data is sent back to the MemCtrl_E1 and MemCtrl_E2 (to keep the figure simple, we omitted the reverse arrows bringing the data back to the CPU). Thus the pipeline path traversed by the example load operation is:

[3]Note that we do not modify the processor instruction set, but rather represent explicitly operations which are implicit in the processor and memory subsystem behavior.

```
(PIPELINE PG, PS, PR, PW, DP, DC, D1_E1, D1_E2, MemCtrl_E1,
     L1_S1, L1_S2, MemCtrl_E1, MemCtrl_E2)
```

Even though this example pipeline path is flattened, the pipeline paths in EXPRESSION are described in a hierarchical manner. In case of an L1 miss, the data request is redirected from L1_S1 to the L2 cache controller, as shown by the pipeline path (the bold arrow) to L2 in Figure 2.

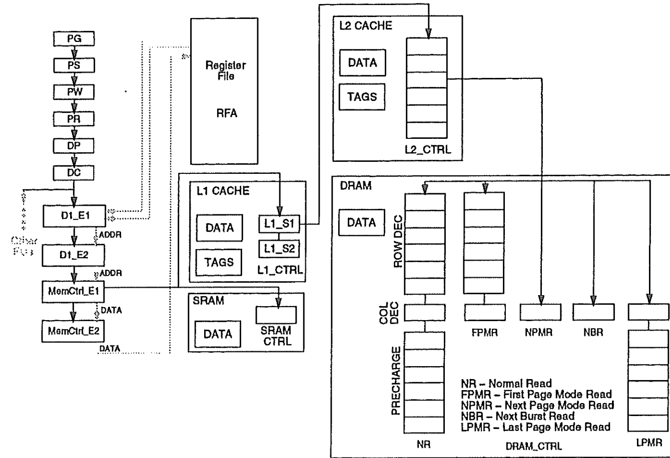

Figure 2. Sample Memory Architecture for the TIC6211

The L2 cache is 4-way set associative, with a size of 1024 lines, and line size of 8 words. The L2 cache controller is non-pipelined, with a latency of 6 cycles:

```
(L2_CTRL (LATENCY 6))
```

During the third cycle of the L2 cache controller, if a miss is detected it is sent to the off-chip DRAM. The DRAM module is composed of the DRAM data block and the DRAM controller, and supports normal, page-mode and burst-mode accesses. A normal access starts with a row decode, where the row part of the address is used to select a particular row from the data array, and copy it into the row buffer. During the column decode, the column part of the address is used to select a particular element from the row buffer and output it. During the precharge, the bank is deactivated. In a page-mode access, if the next access is to the same row, the data can be fetched directly form the row buffer, omitting the column decode and precharge operations. During a burst access, consecutive elements from the row buffer are clocked out on consecutive cycles. Both page-mode and burst-mode accesses, when exploited judiciously generate substantial performance improvements [7]. The timings of each such access mode is represented using the pipeline paths and LATENCY constructs. For instance, the normal read access (NR), composed of a column decode, a row decode and a precharge, is represented by the pipeline path:

```
(PIPELINE COL_DEC ROW_DEC PRECHARGE)
...
(COL_DEC (LATENCY 6))
(ROW_DEC (LATENCY 1))
(PRECHAREGE (LATENCY 6))
```

where the latency of the COL_DEC is 6 cycles, of ROW_DEC is 1 cycle, and of the PRECHARGE is 6 cycles.

In this manner EXPRESSION can model a variety of memory modules and their characteristics. A unique feature of EXPRESSION is the ability to model the *parallelism* and *pipelining* available in and between the memory modules, such as number of outstanding hits, misses or parallel loads, and generate timing and resource information to allow aggressive scheduling to hide the latency of the lengthy memory operations. Parallelism within a module is specified in terms of number of parallel read/writes etc. The parallelism between modules is captured at a higher level e.g., while cache hierarchy is active, DMA can continue to access data from DRAM. EXPRESSION description can be used to drive the generation of both a memory-aware compiler [7], [8], and cycle-accurate structural memory subsystem simulator, and thus enable Design Space Exploration and co-design of the memory and processor architecture.

The EXPRESSION description of the C6x memory subsystem is shown below.

```
(STORAGE_SECTION
        (ScratchPad
                (TYPE SRAM)
                (SIZE 16384)
                (WIDTH 16)
                (NUM_BANKS 2)
                (READ_LATENCY 1)
                (WRITE_LATENCY 1)
                (NUM_PARALLEL_READ 2)
                (NUM_PARALLEL_WRITE 2)
                (READ_WRITE_CONFLICT 1)
                (ADDRESS_RANGE 0 32767)
        )
        (L1
               (TYPE DCACHE)
               (WORDSIZE 32)
               (LINESIZE 8)
               (NUM_LINES 64)
               (ASSOCIATIVITY 2)
               (REPLACEMENT_POLICY LRU)
               (WRITE_POLICY WRITE_BACK)
               (LEVEL 1)
               (NEXT_LEVEL 2)
        )
        (L2
               (TYPE DCACHE)
               (WORDSIZE 32)
               (LINESIZE 8)
               (NUM_LINES 1024)
               (ASSOCIATIVITY 1)         ; Direct Cache
               (REPLACEMENT_POLICY LRU)
               (WRITE_POLICY WRITE_BACK)
               (LEVEL 2)
               (NEXT_LEVEL 3)
        )
        (MainMemeory
                (TYPE DRAM)
                (SIZE 4M)
                (WIDTH 8)
                (LEVEL 3)
                (PIPELINE COL_DEC ROW_DEC PRECHARGE)
        )
        (Connect
                (Type CONNECTIVITY)
```

```
                (STORAGE_CONNECTIONS (L1 L2) (L2 MainMemory))
        )
)
```

6 Memory Subsystem Simulator Generation

In this section we explain the automatic memory subsystem simulator generation process. The memory subsystem is specified in EXPRESSION. Figure 3 shows a basic memory subsystem that we will use as a running example to explain the system implementation. how memory subsystem of the SIMPRESS simulator gets automatically generated from the memory subsystem specification in EXPRESSION. The processor interacts with the memory subsystem through the load/store unit or memory controller. The Load/Store unit issues read, write, prefetch, flush commands. The memory subsystem consists of a memory hierarchy of caches, SRAM, DRAM, stream buffers, victim buffers etc. Once load/store unit issues a read request, the memory subsystem performs the read but delivers the data only in the clock cycle when data is supposed to be available to the processor. ReadQ in the memory subsystem keeps track of the pending read requests. WriteQ keeps track of pending write requests.

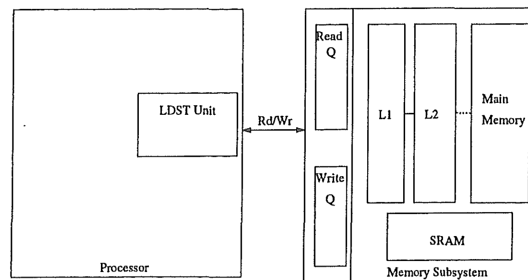

Figure 3. Memory Subsystem

The Load/Store Unit sends read/write request to the memory subsystem. The memory subsystem returns the data after the number of cycles equal to read latency of that read operation. Each storage element SRAM/DRAM/CACHE is modeled in terms of the specified parameters. Interconnection is specified in terms of connections among storage elements. The instantiation of generic modules with appropriate parameters and their connectivity is generated automatically from the EXPRESSION description. For example, the following two functions are generated from the C6x EXPRESSION description discussed in previous section. The first function instantiates the memory modules with appropriate parameters and establishes the connections in the memory hierarchy. The second function determines whom to access for a given address, i.e., whether to look for the data in the cache hierarchy or in the SRAM.

```
//Initialize the memory modules
void Connectivity::initialize()
{
  // L1: <Num_Lines, linesize, associativity, wordsize(bytes), latency>
  MemoryModules[n_modules++]=(MemoryModule*)new AssociativeCache<64, 8, 2, 4, 1>();
```

```
  // L2: <Num_Lines, linesize, wordsize(bytes), latency>
  MemoryModules[n_modules++]=(MemoryModule*)new DirectCache<1024, 8, 4, 4>();

  // MainMemory: <latency>
  MemoryModules[n_modules++]=(MemoryModule*)new Dram<10>();

  // ScratchPad: <latency>
  MemoryModules[n_modules++]=(MemoryModule*)new Sram<1>();

  //Initialize the connections between modules
  addConnection(0,1); // L1 -> L2
  addConnection(1,2); // L2 -> MainMemory
}

// Returns the memory module corresponding to an address
MemoryModule* Connectivity::get_module_for_address(long addr)
{
  if(addr < 32767) {   // 32K --> 0 - 32767
    //the sram
    return MemoryModules[3];
  }
  else{
    //the assoc cache
    return MemoryModules[0];
  }
  return NULL;
}
```

The Load/Store unit sends read/write request to memory subsystem. These requests get stored in the read/write queue in the memory subsystem. The parameter READ_QUEUE_SIZE determines how many read request can be pending in the read queue at a point of time. Similarly the length of the write queue can be controlled by the parameter WRITE_QUEUE_SIZE. These parameters are defined in *MemDefines.h* file. Note that some others parameters e.g., NUM_PARALLEL_READ, NUM_PARALLEL_WRITE, NUM_BANKS, READ_WRITE_CONFLICT , are also defined in the same file.

The memory subsystem performs read/write operation and computes the latency. In case of a miss at any level of the memory hierarchy it initiates refill and applies the replacement policy specified in the EXPRESSION description. Read/write latency is computed using the specified read/write latency of each module accessed for read/write plus refill time in case of a miss. Every clock cycle this latency gets decremented and the Load/Store unit gets the data when latency becomes 0. The Load/Store Unit should perform the following actions every cycle to make use of memory subsystem.

```
extern MemorySubsystem* memSubsystem;

LoadStoreUnit::doStep()
{
```

```
    memSubsystem->preprocess();
    .....
    // Send Read/Write requests -- Val value, long address, int tempOpcode
    // tempOpcode is generic opcode type which specifies what kind of
    // read/write.
    if read operation then
         SendReadRequest(address, tempOpcode);
    else
         SendWriteRequest(value, address, tempOpcode);
    ......
    // Allow memory subsystem to perform read/write operation.
    memSubsystem->doStep();
    ......
    // Check if any of the read is completed. Iterate depending on how
    // many parallel read is allowed.
    int completed = ReceiveData(address, value);
    if (completed == 1)
    {
       // 'value' contains the value read.
    }
}
```

The memory subsystem implementation is correct at the boundary in terms of clock cycles when it delivers the data but internally it is not cycle-accurate. We currently assume that the address for read/write to be a physical address. It may be necessary to perform some changes in the processor side (in terms of control) to be able to use the memory subsystem. Consider the case of modeling the MIPS R10000 memory subsystem. After the address calculation is done (virtual address), the load/store unit (with the help of address queue) issues read request to memory while the TLB is performing virtual to the physical address computation. In the next cycle physical address is ready and data and tags from both the cache-way are compared to determine a hit. At this point of time data is ready (in case of a hit). In case of a miss, refill from secondary to primary memory gets initiated by address queue. The current memory subsystem however, will not be able to model this exact behavior, since separate reading of datas and tags from all the cache-ways are not supported. Instead, one read request can be generated after the physical address computation. In this case data should be available immediately from the memory subsystem (latency 0) to ensure timing correctness at the memory subsystem boundary. Note that in the original case, read latency was one cycle. In both cases final data is available at the same time to the load/store unit.

7 Experiments

As described earlier, we have already used this Memory-Aware Architectural Description Language (ADL) approach to generate a Memory-Aware Compiler [7] and manage the memory miss traffic [8], resulting in significantly improved performance. In this section we demonstrate further use of the memory subsystem specification to describe different configurations of the memory subsystem with the goal of studying the trade-off between cost and performance.

7.1 Experimental Setup

We performed a set of experiments starting from the base TIC6211 processor architecture, and varied the memory subsystem architecture. We generated a cycle-accurate structural memory subsystem simulator, and performed Design Space Exploration of the memory subsystem.

The memory organization of the TIC6211 is varied by using an L1 cache, L2 cache, an off-chip DRAM module, an on-chip SRAM module and a stream buffer module [23].

The L1 cache is a 2-way set associative cache with line size of 4 words and word of 4 bytes. The L2 cache shares a total of 2K on-chip SRAM memory with the direct mapped on-chip SRAM.

The Stream Buffer [23] is used as a replacement for the L2 cache, exhibiting a much smaller data storage size, and slightly more complex control mechanism. It receives a sequence of miss addresses from L1, storing them into a small history buffer. When it recognizes a stream, it allocates one of several FIFO queues to start prefetching it from the DRAM. The stream buffer we implemented contains 4 such FIFO queues, storing 4 cache lines each, and uses an LRU policy to discard a FIFO in case of conflict. When the stream buffer receives an L1 cache miss address, it compares it to the top of each FIFO queue. If the address is found - a stream buffer hit - it pops it from the FIFO and returns it to the L1 cache. If the address is not found - a miss - the stream buffer compares it with the addresses in the history buffer to check for a stream, and sends a request to the DRAM to bring the data.

We used a set of benchmarks from the multimedia and DSP domains, and compiled them using the EXPRESS compiler. We collected the statistics information using the SIMPRESS cycle-accurate structural simulator, which models both the TI6211 processor and the memory subsystem.

The configurations we experimented with are presented in Table 3.

The numbers in Table 3 represent: the size of the memory module (e.g., the size of L1 in configuration 1 is 128), the cache/stream buffer organizations:

```
num_lines x num_ways x line_size x word_size
```

the latency (in number of processor cycles), and the replacement policy (LRU or FIFO). Note that for the stream buffer, num_ways represents the number of FIFO queues present.

7.2 Cost Estimation of the memory configurations

The configurations in Table 3 are presented in increasing order of the cost in terms of area. For the assessment of the actual silicon area occupied by the global background memory, the model presented in [26] is employed. Unlike simpler models proposed in the past (see the references in [26]), which have an acceptable accuracy only for relatively large memories, the Mulder model incorporates the overhead area, like drivers, sense amplifiers, address decoder, and control logic. If the parameters are carefully tuned in advance, this area model has proven to be suitable for comparing the size of (small- to medium-size) on-chip memories of different organizations, yielding no more than 10% error when verified against real memories [26].

According to this model, the actual silicon area occupied by a RAM is given by the formula:

Config	L1 Cache	L2 Cache	SRAM	Stream Buffer	DRAM
1	128B (4x2x4x4) latency=1 cycle (LRU)	-	-	256 (4x4x4x4) latency=4 cycle	latency=20 cycle
2	128B (4x2x4x4) latency=1 cycle (LRU)	-	2K lat=1 cycle	-	latency=20 cycle
3	128B (4x2x4x4) latency=1 cycle (FIFO)	2K (16x4x8x4) latency=4 cycle (FIFO)	-	-	latency=20 cycle
4	128B (4x2x4x4) latency=1 cycle (LRU)	2K (16x4x8x4) latency=4 cycle (LRU)	-	-	latency=20 cycle
5	128B (4x2x4x4) latency=1 cycle (FIFO)	1K (32x1x8x4) latency=4 cycle (FIFO)	1K lat=1 cycle	-	latency=20 cycle
6	-	-	8K lat=1 cycle	-	latency=20 cycle

Table 3. The memory subsystem configurations

$A = TechnoFactor.bits.(1+\alpha Ports).(N+\beta).[1+0.25(Ports+Ports_{rw}-2)]$
where:
$TechnoFactor$ is a technology scaling factor, equal to $(min_geometry[\mu]/2)^2$;
$bits$ - the width of a memory location in bits;
N - the number of storage locations;
$Ports$ - the total number of ports (read, write, and read/write);
$Ports_{rw}$ - the number of read/write ports;
α and β - constants;
The memory configurations we have used here have all the parameters same except N, the number of storage locations. In other words the cost of a configuration is proportional to N. In the cost analysis for the configurations we have considered also the area for the control logic (not only area of the storage locations).

The first configuration contains an L1 cache and a small stream buffer (256 bytes) to capitalize on the stream nature of the benchmarks. The second configuration contains the L1 cache and an on-chip direct mapped SRAM of 2K. A part of the arrays in the application are mapped to the SRAM. Due to the reduced control necessary for the SRAM, it has a small latency (of 1 cycle), and the area requirements are small. The third configuration contains L1 and L2 caches with FIFO replacement policy. Total number of storage locations are exactly same for configuration 2 and configuration 3 but due to the control necessary for the L2 cache (of size 2k), the cost of configuration 3 is larger than the configuration 2 containing the SRAM. Configuration 4 is the same as configuration 3, but with LRU replacement policy for the L1 and L2 caches. Due to the more complex control required by the LRU policy, the cost of this configuration is larger than configuration 3. Configuration 5 contains an L1 cache, an L2 cache of size 1K and a direct mapped SRAM of size 1K. Due to the extra busses to route the data to the caches and SRAM, this configuration has a larger cost than the

previous one. The last configuration contains a large SRAM, and the caches, and has the largest area requirement. All the configurations contain the same off-chip DRAM module with a latency of 20 cycles.

7.3 Results

Figure 4 presents a subset of experiments we ran, showing the total cycle counts (including the time spent in the processor) for the set of benchmarks for different memory configurations attached to the TIC6211 processor. From the experiments we performed, we chose a representative set of benchmarks, which show the different trends in the cost versus performance trade-off. Even though these benchmarks are kernels, we observed a significant variation in the trends shown by the different applications.

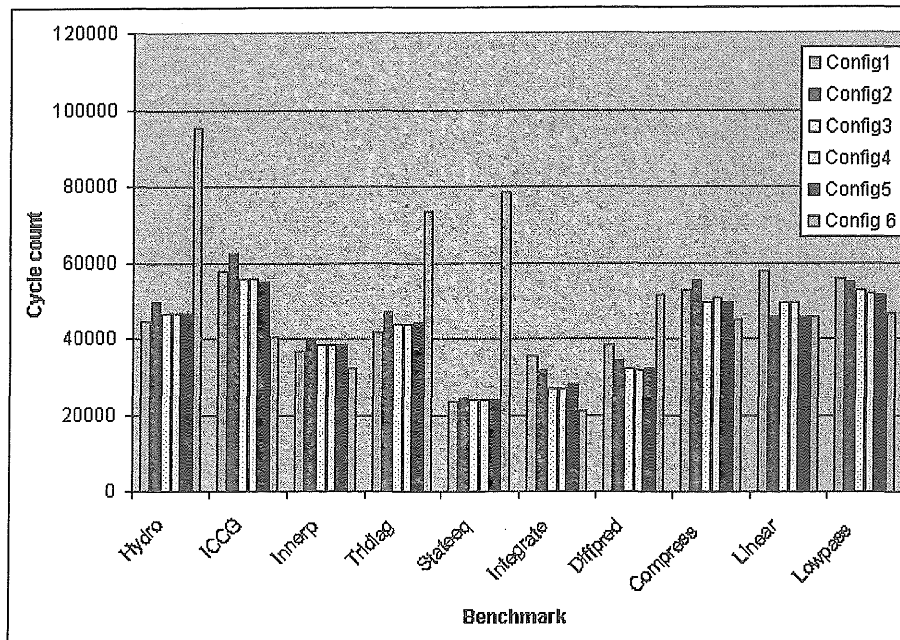

Figure 4. Cycle counts for the memory configurations on TIC6211

For instance, in Hydro, Tridiag, and Stateeq, the first configuration, lowest cost configuration, shows best performance (lower cycle count means higher performance), due to the capability of the stream buffer to exploit efficiently the stream nature of the access patterns. Moreover, in these applications the most expensive configuration (configuration 6), containing the large SRAM behaves poorly, due to the fact that not all the arrays fit in the SRAM, and the lack of L1 cache to compensate the large latency of the DRAM creates its toll on the performance.

The expected trend of higher cost - higher performance was apparent in the applications ICCG, Integrate, and Lowpass, While the stream buffer in configuration 1 has a comparable performance to the other configurations, the configuration 6 has the best behavior due to the low latency of the direct mapped on-chip SRAM.

Thus, using our Memory-Aware ADL-based Design Space Exploration approach, we obtained design points with varying cost and performance. We observed various trends for different application classes, allowing customization of the memory architecture tuned to the applications. Note that this cannot be determined through analysis alone; the customized memory subsystem must be explicitly captured, and the applications have to be executed on the configured processor-memory system, as we demonstrated in this section.

Figure 5, 6, 7, 8, 9, and 10 show the distribution of latency spent in different parts of the processor-memory system for the different configurations shown in Table 3. For each benchmark, the bars show the percentage of time spent in processor(non-load/store) operations, store operations, L1 loads, Stream Buffer loads, L2 loads, SRAM loads, and DRAM loads.

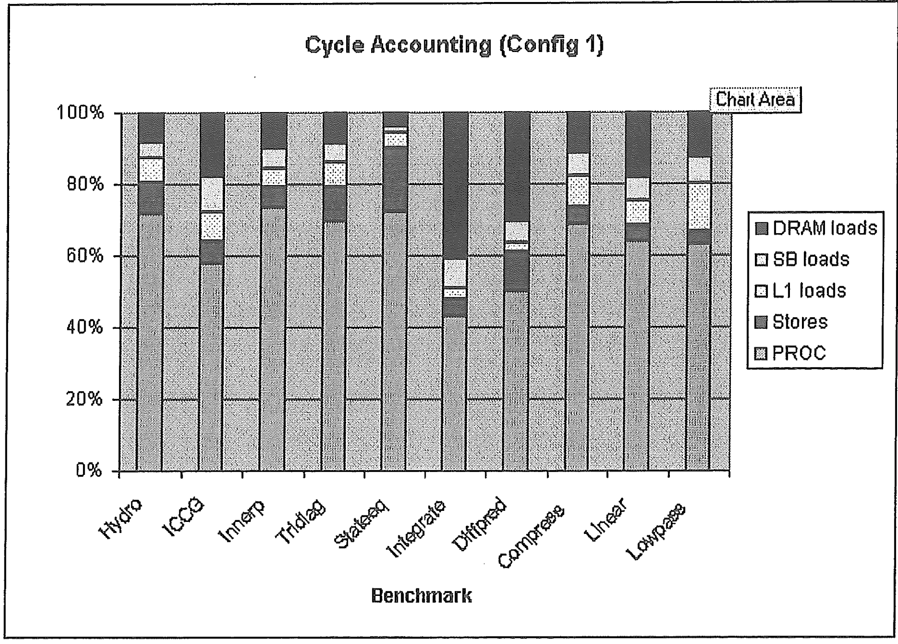

Figure 5. Cycle accounting for memory configuration 1

As mentioned earlier, the performance for configuration 6 is worst for the benchmarks Hydro, Tridiag, and Stateeq whereas other configurations perform equally well on them (Figure 4). Figure 10 shows that for these three benchmarks most of the time(20 - 40%) is spent in DRAM loads whereas for other configurations the time spent for DRAM loads is much lower, 5 -10%. This is due to the fact that not all the arrays fit in the SRAM for configuration 6, and the lack of L1 cache to compensate the large latency of the DRAM creates its toll on the performance.

Configuration 1 has worst performance (see Figure 4) for the benchmark *Linear* inspite of the success of stream buffer (see Figure 5) in detecting and prefetching streams. Due to the small size of the input data set, most of the loads are covered either by SRAM or L1 for other configurations and generated very few DRAM loads as compared to configuration 1.

The accurate analysis of the design space exploration results (Figure 4) is possible due to the detailed cycle accounting information available in Figure 5, 6, 7, 8, 9, and 10. The automatic gen-

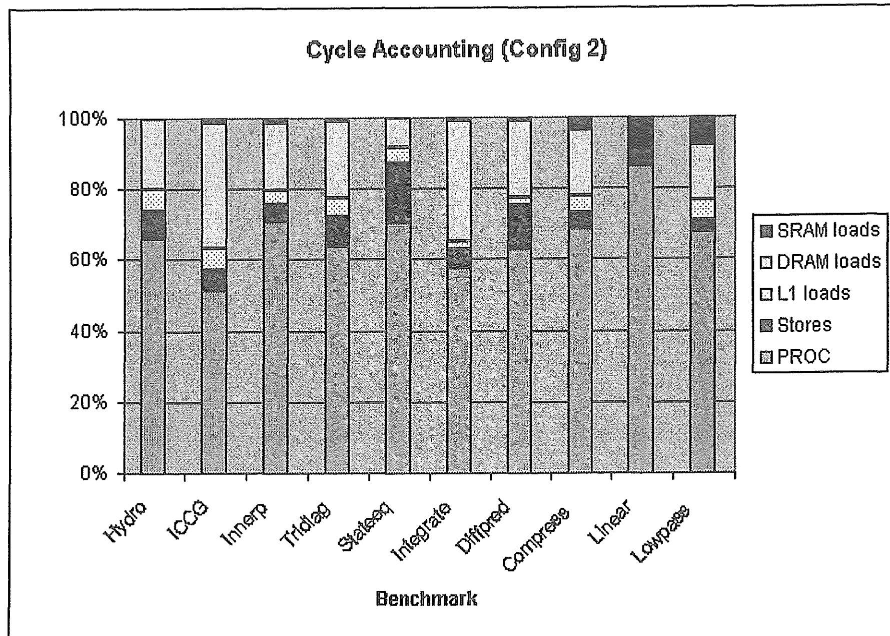

Figure 6. Cycle accounting for memory configuration 2

eration of these detailed information during rapid design space exploration of memory subsystem is possible due to our ADL driven approach.

8 Conclusion

Memory represents a critical driver in terms of cost, performance and power for embedded systems. To address this problem, a large variety of modern memory technologies, and heterogeneous memory organizations have been proposed.

On one hand the application is characterized by a variety of access patterns (such as stream, locality-based, etc.). On the other hand, new memory modules and organizations provide a set of features which exploit specific applications needs (e.g., caches, stream buffers, page-mode, burst-mode, DMA). To find the best match between the application characteristics and the memory organization features, the designer needs to explore different memory configurations in combination with different processor architectures, and evaluate each such system for a set of metrics (such as cost, power, performance). Performing such processor-memory co-exploration requires the capability to capture the memory subsystems.

In this report we presented a Memory-Aware Architectural Description Language (ADL) approach which captures the memory subsystem explicitly.

This Memory-Aware ADL approach is used to drive the generation of a cycle accurate memory simulator, and also facilitate the exploration of various memory configurations, and trade-off cost versus performance. Our experimental results show that varying price-performance design points can be uncovered using the processor-memory co-exploration approach.

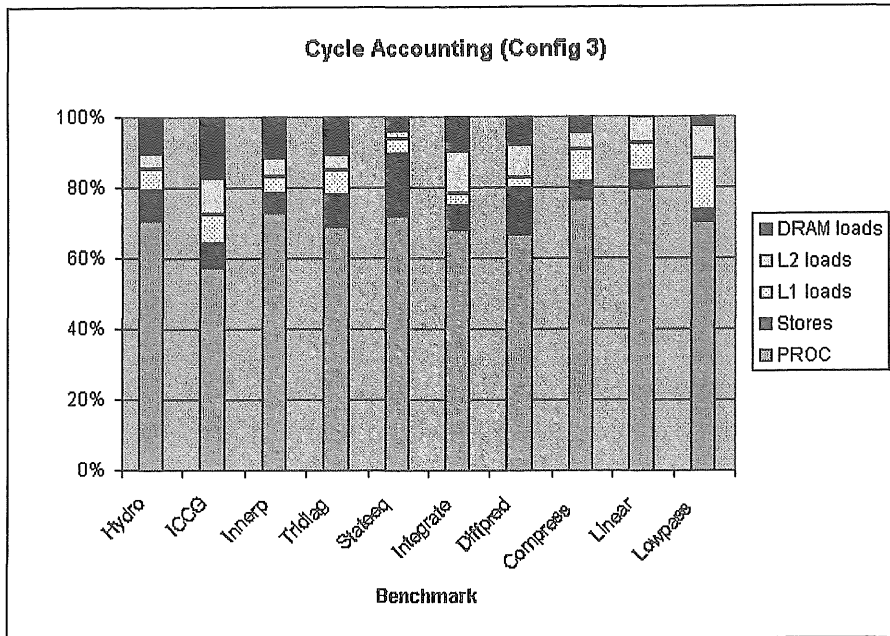

Figure 7. Cycle accounting for memory configuration 3

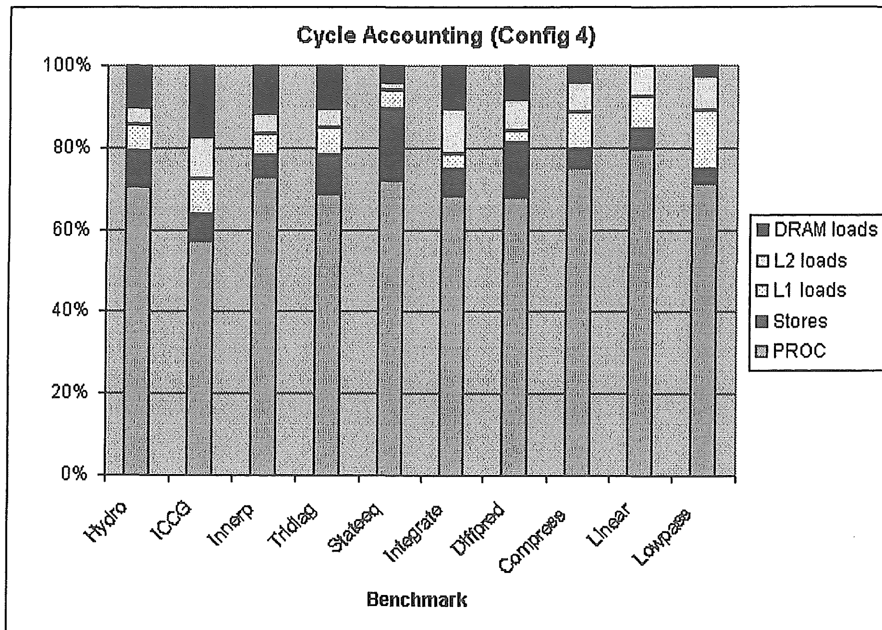

Figure 8. Cycle accounting for memory configuration 4

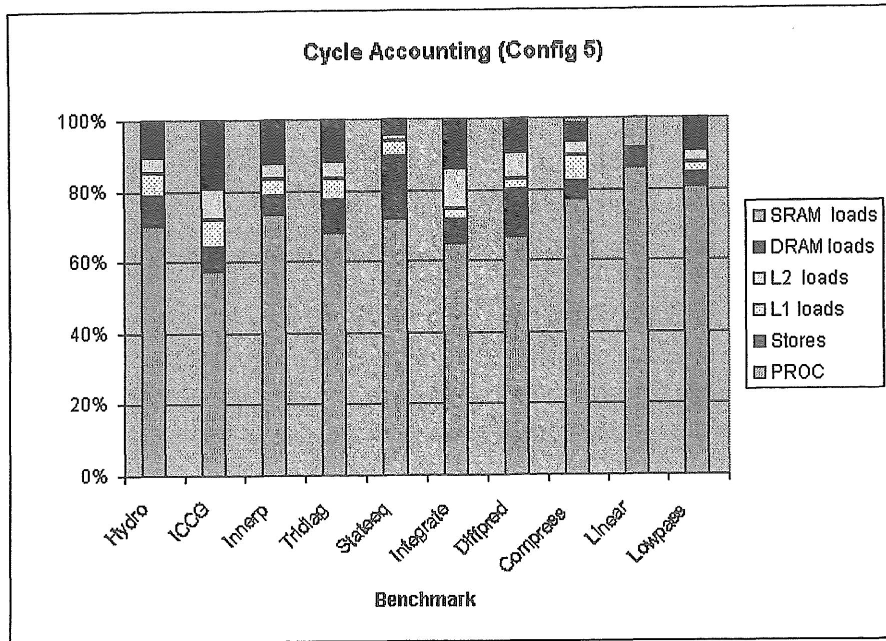

Figure 9. Cycle accounting for memory configuration 5

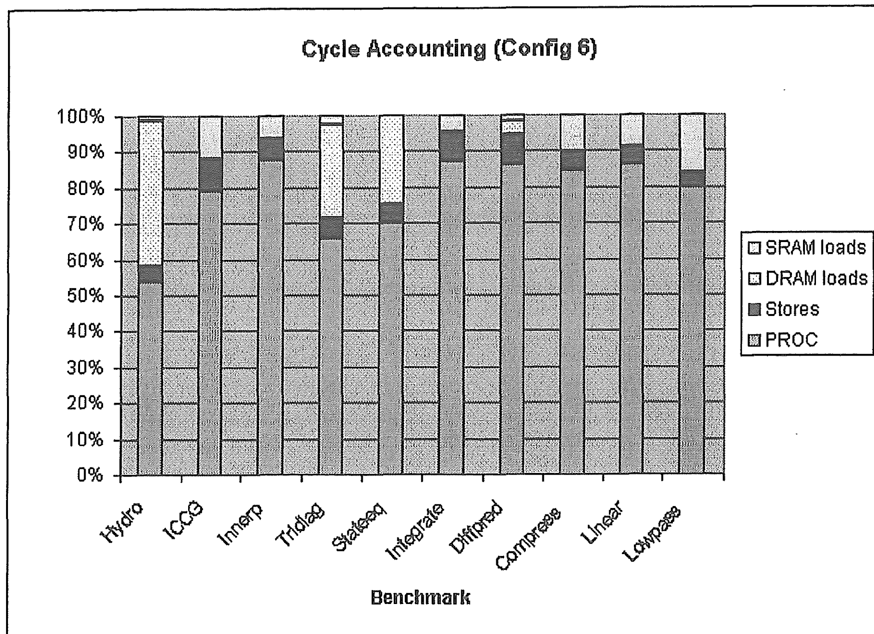

Figure 10. Cycle accounting for memory configuration 6

Our ongoing work targets the use of this ADL approach for further memory exploration experiments, using larger applications, to study the impact of different parts of the application (such as loop nests) on the memory organization behavior and overall performance, as well as on system power.

The memory subsystem for the cycle-accurate retargetable simulator is automatically generated from the EXPRESSION description. The memory subsystem is modeled using EXPRESSION description for C6x, R10K, IA-64 and PowerPC architectures. The automatic generation of memory subsystem from EXPRESSION description is tested on livermoore loop and multimedia kernels for C6x, R10K and PowerPC processors. Endianness is not supported in the current version of the memory subsystem simulator implementation and is subject of on-going work.

9 Acknowledgements

This work was partially supported by grants from NSF (MIP-9708067), DARPA (F33615-00-C-1632) and a Motorola fellowship. We would like to gratefully acknowledge Ashok Halambi, Srikanth Srinivasan, Soumitri Pal, Nick Saviou and all other EXPRESSION team members for their contribution to the memory subsystem exploration work.

References

[1] ARC Cores. *http://www.arccores.com.*

[2] J. S. O. Chris Basoglu, Woobin Lee. *The MAP1000A VLIW Mediaprocessor*, 2000.

[3] G. G. et al. CHESS: Retargetable code generation for embedded DSP processors. In *Code Generation for Embedded Processors.* Kluwer, 1997.

[4] G. H. et al. ISDL: An instruction set description language for retargetability. In *Proc. DAC*, 1997.

[5] R. L. et al. Retargetable generation of code selectors from HDL processor models. In *Proc. EDTC*, 1997.

[6] M. Freericks. The nML machine description formalism. Technical Report TR SM-IMP/DIST/08, TU Berlin CS Dept., 1993.

[7] P. Grun, N. Dutt, and A. Nicolau. Memory aware compilation through accurate timing extraction. In *DAC*, Los Angeles, 2000.

[8] P. Grun, N. Dutt, and A. Nicolau. Mist: An algorithm for memory miss traffic management. In *To Appear in ICCAD*, San Jose, 2000.

[9] P. Grun, A. Halambi, N. Dutt, and A. Nicolau. RTGEN: An algorithm for automatic generation of reservation tables from architectural descriptions. In *ISSS*, San Jose, CA, 1999.

[10] A. Halambi, P. Grun, V. Ganesh, A. Khare, N. Dutt, and A. Nicolau. EXPRESSION: A language for architecture exploration through compiler/simulator retargetability. In *Proc. DATE*, Mar. 1999.

[11] http://developer.intel.com/design/ia-64/architecture.htm. *IA-64 Architecture.*

[12] http://developer.intel.com/design/strong/sa1100.htm. *StrongARM Processors.*

[13] http://ebus.mot-sps.com/ProdCat/psp/0,1250,DSP56002~M98595,00.html. *DSP56002: 24-bit Digital Signal Processor.*

[14] http://www.chipanalyst.com/pc-processors/nec-v830. *NEC V830 Adds Speed, New Instructions.*

[15] http://www.ia-64.hp.com/parisc/8500-wp.html. *PA-8500: The Continuing Evolution of the PA-8000 Family.*

[16] http://www.lucent.com/micro/Starcore. *Starcore, Next Generation DSPs.*

[17] http://www.motorola.com/SPS/PowerPC. *MPC7400 PowerPC Microprocessor.*

[18] http://www.motorola.com/SPS/PowerPC/Altivec. *Altivec: Motorola's high-performance vector parallel processing expansion to the PowerPC^{TM} architecture.*

[19] http://www.qedinc.com/5720.htm. *QED Alpine RM57x.*

[20] http://www.sgi.com/processors/r10k. *MIPS R10000 Microprocessor.*

[21] http://www.sun.com/microelectronics/UltraSparc-III. *UltraSparc III.*

[22] http://www.ti.com/sc/docs/products/dsp/C6000/index.htm. *TMS320C6000^{TM} Highest Performance DSP Platform.*

[23] N. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *ISCA*, 1990.

[24] A. Khare, N. Savoiu, A. Halambi, P. Grun, N. Dutt, and A. Nicolau. V-SAT: A visual specification and analysis tool for system-on-chip exploration. In *Proc. EUROMICRO*, 1999.

[25] Motorola Inc. *DSP56000 Digital Signal Processing Family Manual, 1992.*

[26] J. M. Mulder, N. T. Quach, and M. J. Flynn. An area model for on-chip memories and its application. *IEEE Journal of Solid State Circuits*, SC-26(1):98–105, Feb 1991.

[27] V. Rajesh and R. Moona. Processor modeling for hardware software codesign. In *International Conference on VLSI Design*, Jan. 1999.

[28] Tensilica Incorporated. *http://www.tensilica.com.*

[29] Texas Instruments. *TMS320C5x General-Purpose Applications User's Guide*, 1997.

[30] Trimaran Release: http://www.trimaran.org. *The MDES User Manual*, 1997.