Processor-Memory Coexploration Using an Architecture Description Language

PRABHAT MISHRA, MAHESH MAMIDIPAKA, and NIKIL DUTT
University of California, Irvine

Memory represents a major bottleneck in modern embedded systems in terms of cost, power, and performance. Traditionally, memory organizations for programmable embedded systems assume 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 specific target applications. However, such a processor–memory coexploration approach critically needs the ability to explicitly capture heterogeneous memory architectures. We present in this paper a language-based approach to explicitly capture the memory subsystem configuration, generate a memory-aware software toolkit, and perform coexploration of the processor–memory architectures. We present a set of experiments using our memory-aware architectural description language (ADL) to drive the exploration of the memory subsystem for the TI C6211 processor architecture, demonstrating cost, performance, and energy trade-offs.

Categories and Subject Descriptors: C.0 [**Computer Systems Organization**]: General—*Modeling of Computer Architecture*; C.0 [**Computer Systems Organization**]: General—*Systems Specification Methodology*; C.3 [**Computer Systems Organization**]: Special-Purpose and Application-based Systems—*Real-time and Embedded Systems*; C.4 [**Computer Systems Organization**]: Performance of Systems—*Modeling Techniques*

General Terms: Design, Experimentation, Languages

Additional Key Words and Phrases: Processor–memory codesign, memory exploration, architecture description language, design space exploration

1. INTRODUCTION

Memory represents a major cost, power, and performance bottleneck for a large class of embedded systems. Thus, system designers pay great attention to the design and tuning of the memory architecture early in the design process. However, not many system-level tools exist to help the system designers evaluate the effects of novel memory architectures, and facilitate simultaneous exploration of the processor and memory subsystem.

While a traditional memory architecture for programmable systems was organized as a cache hierarchy, the widening processor/memory performance gap [SIA 1998] requires more aggressive use of memory configurations, *customized* for the specific target applications. To address this problem, recent advances

Authors' address: Center for Embedded Computer Systems, 444 Computer Science, University of California, Irvine, CA 92697-3430; email: pmishra@ics.uci.edu.

in memory technology have generated a plethora of new and efficient memory modules (e.g., SDRAM, DDRAM, and RAMBUS), exhibiting a heterogeneous set of features (e.g., page-mode, burst-mode and pipelined accesses).

On the other hand, many embedded applications exhibit varied memory access patterns that naturally map into a range of heterogeneous memory configurations (containing for instance multiple cache hierarchies, stream buffers, on-chip and off-chip direct-mapped memories). In the design of traditional programmable systems, the processor architect typically assumed a fixed cache hierarchy, and spent significant amount of time optimizing the processor architecture; thus the memory architecture is implicitly fixed and optimized separately from the processor architecture.

Due to the heterogeneity in recent memory organizations and modules, there is a critical need to address the memory-related optimizations simultaneously with the processor architecture and the target application. Through coexploration of the processor and the memory architecture, it is possible to exploit the heterogeneity in the memory subsystem organizations, and trade-off system attributes such as cost, performance, and power. However, such processor–memory coexploration framework requires the capability to explicitly capture, exploit, and refine both the processor as well as the memory architecture.

The contribution of this paper is the explicit description of a customized, heterogeneous memory architecture in our EXPRESSION ADL [Halambi et al. 1999], permitting coexploration of the processor and the memory architecture. By viewing the memory subsystem as a “first class object,” we generate a memory-aware software toolkit (compiler and simulator), and allow for memory-aware design space exploration (DSE).

The rest of the paper is organized as follows. Section 2 presents related work addressing architecture description language (ADL)-driven DSE approaches. Section 3 outlines our approach and the overall flow of our environment. Section 4 presents a simple example to illustrate how a compiler can exploit memory subsystem description. Section 5 presents the memory subsystem description in EXPRESSION, followed by a contemporary example architecture in Section 6. Section 7 presents memory architecture exploration experiments using the TIC6211 processor, with varying memory configurations to explore design points for cost, energy and performance. Section 8 concludes the paper.

2. RELATED WORK

We discuss related research in two categories. First, we survey recent approaches on ADL driven DSE, and second, we discuss previous works on embedded system exploration.

An extensive body of recent research addresses ADL driven software toolkit generation and DSE for processor-based embedded systems, in both academia: ISDL [Hadjiyiannis et al. 1997], Valen-C [Inoue et al. 1998], MIMOLA [Leupers and Marwedel 1997], LISA [Zivojnovic et al. 1996], nML [Freericks 1993], Sim-nML [Rajesh and Moona 1999]; and industry: ARC [ARC], Axys [Axys], RADL [Siska 1998], Target [Target], Tensilica [Tensilica], MDES [Trimaran 1997].

Table I. Specification Changes for Architectural Modifications

	Number of Lines Modified		Number of Sections Modified	
Architectural Modifications	EXPRESSION	MDES	EXPRESSION	MDES
Add a connection	2	5	2	4
Add a bus & two connections	6	17	2	7
Split a register-file	3	18	2	7

While these approaches explicitly capture the processor features to varying degrees (e.g., instruction set, structure, pipelining, resources), to our knowledge, no previous approach has explicit mechanisms for specification of a customized memory architecture that describes the specific types of memory modules (e.g., caches and stream/prefetch buffers), their complex memory features (e.g., page-mode and burst-mode accesses), their detailed timings, resource utilization, and the overall organization of the memory architecture (e.g., multiple cache hierarchies, partitioned memory spaces, and direct-mapped memories).

EXPRESSION is well suited as a language for rapid design space exploration. It allows easy and natural way of specifying architectures. The amount of modification needed in the specification is minimal. For example, the behavior of each instruction in LISA [Zivojnovic et al. 1996] is described using pipeline structure explicitly. Therefore, a slight change in the pipeline structure would require modification of all the instruction behaviors. In EXPRESSION the instruction set description is not affected by the modification of the pipeline structure. Table I shows the amount of modification needed in EXPRESSION and MDES [Trimaran 1997] language to perform three architectural changes. The first column lists the architectural modifications performed. The second and third columns show the number of lines to be modified for the architectural change in EXPRESSION and MDES, respectively. Finally, the fourth and fifth columns presents the number of sections to be modified for the change in EXPRESSION and MDES, respectively.

Memory exploration for embedded systems has been addressed by Panda et al. [1997]. The metric used for the system are data cache size and number of processor cycles. The method has been extended by Shiue and Chakrabarti [1999] to include energy consumption as one of the metric. Catthoor et al. [1998] have presented a methodology for memory hierarchy and data reuse decision exploration. Grun et al. proposed techniques for early memory [Grun et al. 2001] and connectivity [Grun et al. 2002] architecture exploration.

The work by Slock et al. [1997] presents a memory exploration technique based on the data layout and flow graph analysis of the applications. Their work tries to minimize the required memory bandwidth by optimizing the access conflict graph for groups of scalars within a given cycle budget. The work by Lee et al. [1998] presents a framework for exploring programmable processors for a set of applications. They used IMPACT tool suit [Chang et al. 1991] to collect run-times of the benchmarks on different processor configurations by varying processor features. They performed K-selection algorithm to select a set of machine configurations based on area and performance constraints.

A system-level performance analysis and DSE methodology (SPADE) is proposed by Lieverse et al. [1999]. In this methodology, the application tuning

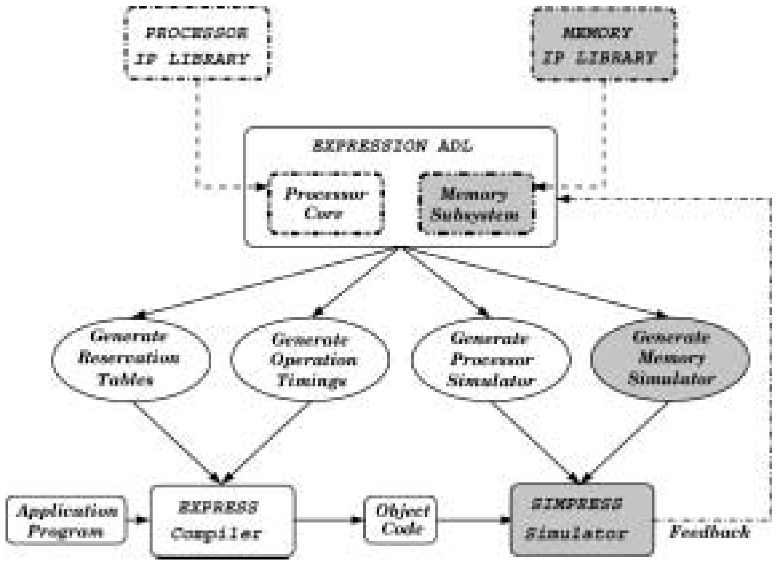


Fig. 1. Processor–memory coexploration framework.

is driven manually by the designer. Several DSE approaches use heuristics to prune the potentially large design space. Givargis et al. [2001] used a clustering-based technique for system-level exploration in which independent parameters are grouped into different clusters. An exhaustive search is performed only on elements within a cluster (i.e., on dependent parameters) thereby reducing the search space. Ascia et al. [2001] proposed a technique to map the exploration problem to a genetic algorithm. Fornaciari et al. [2001] use a sensitivity-based technique in which the sensitivity of each parameter over the design objective is determined using experiments. The exploration is performed on each parameter independently in the order determined by the sensitivities.

These approaches assumed a relatively fixed memory structure. The memory modules considered are traditional cache hierarchies and SRAMs. Also, the compiler is not aware of the memory subsystem. Our framework allows exploration of generic memory configurations consisting of varied connectivity and modules. The memory subsystem exploration is performed along with any processor architecture driven by an ADL. Designers specify the processor and memory subsystem configuration in an ADL as an input to our automatic exploration framework. Any of the exploration algorithms and pruning techniques proposed in the abovementioned approaches can be used to generate the ADL description during design space exploration.

3. OUR APPROACH

Figure 1 shows our processor–memory coexploration framework. In our IP library-based DSE scenario, the designer starts by selecting a set of components from a processor IP library and memory IP library. The EXPRESSION [Halambi et al. 1999] 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 [Mishra et al. 2001].

Our previous work on reservation table [Grun et al. 1999] and operation timing generation [Grun et al. 2000a] algorithms can exploit this detailed timing information to hide the latency of the lengthy memory operations. Section 4 shows an example of performance improvement due to this detailed memory subsystem timing information [Grun et al. 2000a]. Such aggressive optimizations in the presence of efficient memory access modes (e.g., page/burst modes) and cache hierarchies [Grun et al. 2000b] are only possible due to the explicit representation of the detailed memory architecture. We generate memory simulator (shown shaded in Figure 1) that is integrated into the SIMPRESS [Khare et al. 1999] simulator, allowing for detailed feedback on the memory subsystem architecture and its match to the target applications.

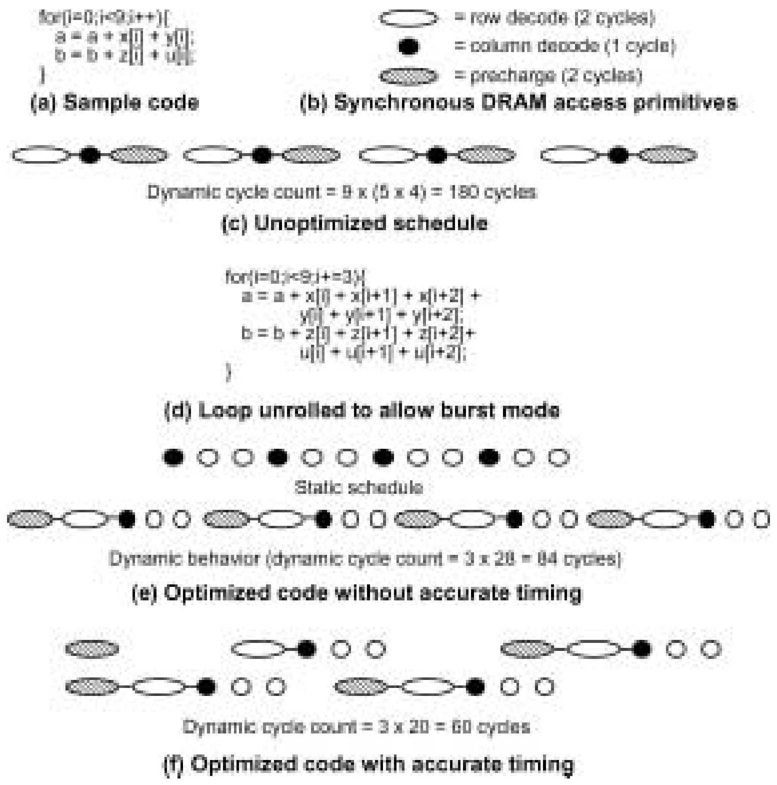
4. MOTIVATING EXAMPLE

A typical access mode for contemporary DRAMs (e.g., SDRAM) is burst-mode access, that is not fully exploited by traditional compilers. This example shows the performance improvement made possible by compiler exploitation of such access modes through a more accurate memory timing model.

The sample memory library module used here is the IBM synchronous DRAM [IBM0316409C]. This memory contains two banks, organized as arrays of 2048 rows × 1024 columns, and supports normal, page-mode, and burst-mode accesses. A normal read access starts by a row decode (activate) stage, where the entire selected row is copied into the row buffer. During column decode, the column address is used to select a particular element from the row buffer and output it. The normal read operation ends with a precharge (or deactivate) stage, wherein the data lines are restored to their original values. For page-mode reads, if the next access is to the same row, the row decode stage can be omitted, and the element can be fetched directly from the row buffer, leading to a significant performance gain. Before accessing another row, the current row needs to be precharged. During a burst-mode read, starting from an initial address input, a number of words equal to the burst length are clocked out on consecutive cycles without having to send the addresses at each cycle.

Another architectural feature that leads to higher bandwidth in this DRAM is the presence of two banks. While one bank is bursting out data, the other can perform a row decode or precharge. Thus, by alternating between the two banks, the row decode and precharge times can be hidden. Traditionally, the architecture would rely on the memory controller to exploit the page/burst access modes, while the compiler would not use the detailed timing model. In our approach, we incorporate accurate timing information into the compiler, which allows the compiler to exploit such parallelism globally, and better hide the latencies of the memory operations.

A sample code shown in Figure 2(a) is used to demonstrate the performance of the system in three cases: (I) without efficient access modes, (II) optimized for burst-mode accesses, but without an accurate timing model, and (III) optimized for burst-mode accesses with an accurate timing model. The primitive



```
for(i=0;i<9;i++){
    a = a + x[i] + y[i];
    b = b + z[i] + u[i];
}
```

```
for(i=0;i<9;i+=3){
    a = a + x[i] + x[i+1] + x[i+2] +
            y[i] + y[i+1] + y[i+2];
    b = b + z[i] + z[i+1] + z[i+2]+
            u[i] + u[i+1] + u[i+2];
}
```

= row decode (2 cycles)
= column decode (1 cycle)
= precharge (2 cycles)
(a) Sample code
(b) Synchronous DRAM access primitives
Dynamic cycle count = 9 x (5 x 4) = 180 cycles
(c) Unoptimized schedule
(d) Loop unrolled to allow burst mode
Static schedule
Dynamic behavior (dynamic cycle count = 3 x 28 = 84 cycles)
(e) Optimized code without accurate timing
Dynamic cycle count = 3 x 20 = 60 cycles
(f) Optimized code with accurate timing

Fig. 2. A motivating example.

access mode operations for a synchronous DRAM are shown in Figure 2(b): the unshaded node represents the row decode operation (taking 2 cycles), the solid node represents the column decode (taking 1 cycle), and the shaded node represents the precharge operation (taking 2 cycles). Figure 2(c) shows the schedule for the unoptimized version, where all reads are normal memory accesses (composed of a row decode, column decode, and precharge). The dynamic cycle count for this case is $9 \times (5 \times 4) = 180$ cycles.

In order to increase the data locality and allow burst-mode access to read consecutive data locations, an optimizing compiler would unroll the loop three times. Figure 2(d) shows the unrolled code. Figure 2(e) shows the static and the dynamic (run-time) schedule of the code[1] for a schedule with no accurate timing. Traditionally, the memory controller would handle all the special access modes implicitly, and the compiler would schedule the code optimistically, assuming that each memory access takes 1 cycle (the length of a page-mode access). During a memory access that takes longer than expected, the memory controller has to freeze the pipeline, to avoid data hazards. Thus, even though the static schedule seems faster, the dynamic cyclecount in this case is $3 \times 28 = 84$ cycles.

[1]In Figure 2(c) the static schedule and the run-time behavior were the same. They are different in this case due to the stalls inserted by the memory controller.

Figure 2(f) shows the effect of scheduling using accurate memory timing on code that has already been optimized for burst mode. As the memory controller does not need to insert stalls anymore, the dynamic schedule is the same as the static one. Since accurate timing is available, the scheduler can hide the latency of the precharge and row decode stages, by precharging the two banks at the same time, or executing row decode while the other bank bursts out data. The dynamic cycle count here is $3 \times 20 = 60$ cycles, resulting in a 40% improvement over the best schedule a traditional optimizing compiler would generate.

Thus, by providing the compiler with more detailed information, the efficient memory access modes can be better exploited. The more accurate timing model creates a significant performance improvement, in addition to the page/burst-mode optimizations. The optimizing compilers have traditionally been designed to exploit special architectural features of the processor (e.g., detailed pipeline information). Traditionally, the memory features (e.g., DRAM access modes) were transparent to the processor, and were exploited implicitly by the memory controller. However, the memory controller only has access to local information, and is unable to perform more global optimizations (such as global code reordering to better exploit special memory access modes). By providing the compiler with a more accurate timing model for the specific memory access modes, it can perform global optimizations to generate a better performance. The compiler can combine the timing model of the memory modules with the processor pipeline timings to generate accurate operating timings. The exact operation timings are used to better schedule the application, and hide the latency of the memory operations [Grun et al. 2000a].

5. THE MEMORY SUBSYSTEM DESCRIPTION IN EXPRESSION

In order to explicitly describe the memory architecture in EXPRESSION, we need to capture both structure and behavior of the memory subsystem. The memory structure refers to the organization of the memory subsystem containing memory modules and the connectivity among them. The behavior refers to the memory subsystem instruction set.

The memory subsystem instruction set represents the possible operations that can occur in the memory subsystem, such as data transfers between different memory modules or to the processor (e.g., load and store), control instructions for the different memory components (such as the DMA), or explicit cache control instructions (e.g., cache freeze, prefetch, replace, and refill).

The memory subsystem structure represents the abstract memory modules (such as caches, stream buffers, RAM modules), their connectivity, and characteristics (e.g., cache properties). The memory subsystem structure is represented as a netlist of memory components connected through ports and connections. The memory components are described and attributed with their characteristics (such as cache line size, replacement policy, write policy).

Table II shows the primitives used in the memory subsystem description. The first column represents the name of the parameter, the second column represents the possible values for that parameter, and the third column provides a brief description of the parameter.

Table II. Memory Subsystem Primitives

Parameter	Values	Description
STORAGE_SECTION		Start of memory description
TYPE	SRAM, DRAM, REGFILE, CONNECTIVITY, DCACHE, WRITE_BUFFER, VICTIM_BUFFER, ICACHE, STREAM_BUFFER	Type of the component
SIZE	positive integer	Number of storage locations
WIDTH	positive integer	Num of bits in each storage
ADDRESS_RANGE	two positive integers	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	Cache replacement policy
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 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	boolean	true means bank R/W conflict
PIPELINE	pipeline stages	describes the pipeline paths
ACCESS_MODES	page, burst, pipelined, and so on	memory access modes

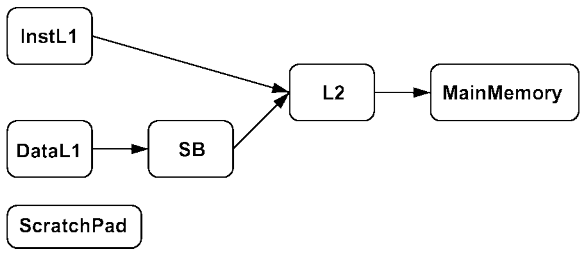


Fig. 3. A simple memory subsystem.

The pipeline stages and parallelism for each memory module, its connections, and ports, as well as the latches between the pipeline stages are described explicitly, to allow modeling of resource and timing conflicts in the pipeline. The semantics of each component is represented in C, as part of a parameterizable components library. We are able to describe the memory subsystem for wide variety of architectures, including RISC, DSP, VLIW, and Superscalar.

The memory subsystem is described within STORAGE_SECTION of the EXPRESSION description. The following sample STORAGE_SECTION describes the memory subsystem shown in Figure 3.

```
(STORAGE_SECTION
   (DataL1
      (TYPE DCACHE)
      (WORDSIZE 64)
      (LINESIZE 8)
      (NUM_LINES 1024)
      (ASSOCIATIVITY 2)
      (REPLACEMENT_POLICY LRU)
      (WRITE_POLICY WRITE_BACK)
      (READ_LATENCY 1)
      ............
   )
   (ScratchPad
      (TYPE SRAM) (ADDRESS_RANGE 0 4095)
      ..........
   )
   (SB
      (TYPE STREAM_BUFFER) ....
   )
   (InstL1
      (TYPE ICACHE) ..........
   )
   (L2
      (TYPE DCACHE) ..........
   )
   (MainMemeory
      (TYPE DRAM) ............
   )
   (Connect
      (TYPE CONNECTIVITY)
      (CONNECTIONS
         (InstL1, L2) (DataL1, SB) (SB, L2) (L2, MainMemory)
      )
   )
)
```

It has separate instruction and data caches (InstL1, DataL1). It has a unified L2 cache. To get the advantage of the streaming nature of the data it uses a stream buffer (SB). It also uses a small (4K) SRAM as a scratch pad data memory. Each of the memory modules have different parameter values. 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.

Section 6 describes how to describe TI C6211 memory subsystem using the primitives described in this section. Further details on the memory subsystem description in EXPRESSION can be found in Mishra et al. [2000].

6. EXAMPLE MEMORY ARCHITECTURE

We illustrate our memory-aware ADL using the Texas Instruments TIC6211 VLIW DSP [Texas Instruments 1998] processor that has several novel memory features. Figure 4 shows the example architecture, containing an off-chip

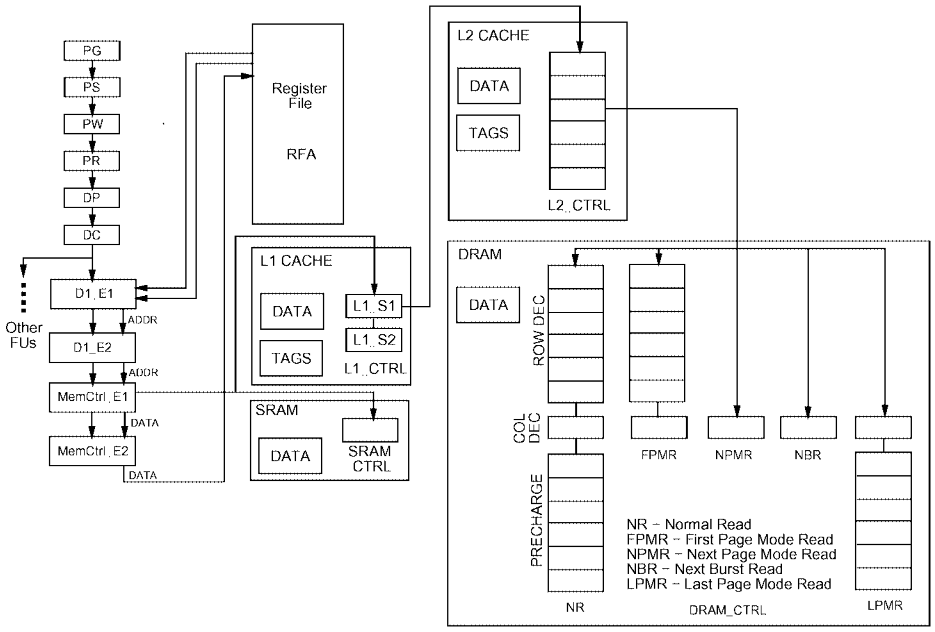


Fig. 4. Sample memory architecture for TIC6211.

DRAM, an on-chip SRAM, and two levels of caches (L1 and L2), attached to the memory controller of the TIC6211 processor. 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 4. TI C6211 is an eight-way VLIW DSP processor with a deep pipeline, composed of four fetch stages (PG, PS, PW, PR), two decode stages (DP, DC), followed by eight functional units. The D1 load/store functional unit pipeline is composed of D1_E1, D1_E2, and the two memory controller stages: MemCtrl_E1 and MemCtrl_E2.

The L1 cache is a two-way set-associative cache, with a size of 64 lines, a line size of four 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 two stages (L1_S1, L1_S2). The cache characteristics are described as part of the STORAGE_SECTION in EXPRESSION:

```
(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 [Halambi et al. 1999]:

```
(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. We do not modify the processor instruction set, rather represent operations explicitly that are implicit in the processor and memory subsystem behavior.

The pipelining and parallelism between the cache operations are described in EXPRESSION through pipeline paths [Halambi et al. 1999]. Pipeline paths represent the ordering between pipeline stages in the architecture (represented as bold arrows in Figure 4). For instance, a load operation to a DRAM address traverses first the four fetch stages (PG, PS, PW, PR) of the processor, followed by the two 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:

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

Even though the pipeline path is flattened in this example, the pipeline paths in EXPRESSION are described in a hierarchical manner. In case of a 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 4.

The L2 cache is four-way set associative, with a size of 1024 lines, and line size of eight words. The L2 cache controller is nonpipelined, with a latency of six 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 the corresponding data 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 [Grun et al. 2000a]. 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 ROW_DEC COL_DEC PRECHARGE)
...
(ROW_DEC (LATENCY 6))
(COL_DEC (LATENCY 1))
(PRECHAREGE (LATENCY 6))
```

where the latency of the row decode (ROW_DEC) is six cycles, column decode (COL_DEC) is one cycle, and of the precharge (PRECHARGE) is six 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 the memory modules. This allows the compiler to generate timing and resource information to allow aggressive scheduling to hide the latency of the lengthy memory operations. The EXPRESSION description can be used to drive the generation of both a memory-aware compiler [Grun et al. 2000a], and cycle-accurate structural memory subsystem simulator, and thus enable coexploration of processor and memory architecture. For more details on the memory subsystem description in EXPRESSION and automatic software toolkit generation refer to Mishra et al. [2000].

7. EXPERIMENTS

As described earlier, we have already used this memory-aware ADL to generate a compiler [Grun et al. 2000a] and manage the memory miss traffic [Grun et al. 2000b], resulting in significantly improved performance. We performed comparative studies with the MULTI integrated development environment (IDE) version 3.5 from Green Hills Software Inc. [2003] for the MIPS R4000 processor. We obtained the evaluation copy of the software.

Table III. Comparison Between EXPRESSION and MULTI IDE

Benchmarks	Code Size (Bytes)		Cycle Count	
	EXPRESSION	MULTI IDE	EXPRESSION	MULTI IDE
Compress	284	252	9,218	8,233
GSR	468	524	14,098	14,580
Laplace	476	876	11,562	12,430
Linear	332	488	7,962	7,273
Lowpass	504	400	10,922	10,170
SOR	656	576	12,300	14,524
Wavelet	300	92	7,609	6,805

Table IV. Benchmarks

Benchmark	Description
Compress	Image compression scheme
GSR	Red-black Gauss-Seidel relaxation method
Hydro	Hydro fragment
DiffPred	Difference predictors
FirstSum	First sum
FirstDiff	First difference
PartPush	2-D PIC (Particle In Cell)
1DPartPush	1-D PIC (Particle In Cell)
CondCompute	Implicit, conditional computation
Hydrodynamics	2-D explicit hydrodynamics fragment
GLRE	General linear recurrence equations
ICCG	ICCG excerpt (Incomplete Cholesky Conjugate Gradient)
MatMult	Matrix multiplication
Planc	Planckian distribution
2DHydro	2-D implicit hydrodynamics fragment
FirstMin	Find location of first minimum in array
InnerProd	Inner product
LinearEqn	Banded linear equations
TriDiag	Tri-diagonal elimination, below diagonal
Recurrence	General linear recurrence equations
StateExcerpt	Equation of state fragment
Integrate	ADI integration
IntPred	Integrate predictors
Laplace	Laplace algorithm to perform edge enhancement
Linear	Implements a general linear recurrence solver
Wavelet	Debaucles 4-Coefficient Wavelet filter

Table III presents the code size and simulation cycle count for the multimedia benchmarks. The description of the benchmarks are given in Table IV. We enabled all the performance optimization switches in our compiler. We enabled the *optimize for speed* switch in MULTI IDE compiler. Both simulators are cycle-accurate model of the MIPS 4000 processor. The first column lists the benchmarks. The second and third columns present the code size generated by the EXPRESSION and MULTI IDE compiler respectively. The last two columns present the cycle count generated by the EXPRESSION and MULTI IDE (MIPSsim) simulators respectively. Our software toolkit has comparable performance with MULTI IDE compiler and simulator.

In this section we demonstrate use of the memory subsystem specification to describe different memory configurations and perform DSE with the goal of evaluating the configurations for cost, power, and performance. We describe the experimental setup, followed by the estimation models used in our framework for performance, area, and energy computations. Finally, we present the results.

7.1 Experimental Setup

We performed a set of experiments starting from the base TI C6211 [Texas Instruments 1998] processor architecture, and varied the memory subsystem architecture. We generated a memory-aware software toolkit (compiler and simulator), and performed DSE of the memory subsystem. The memory organization of the TIC6211 is varied by using separate L1 instruction and data caches, an L2 cache, an off-chip DRAM module, an on-chip SRAM module and a stream buffer module [Jouppi 1990] with varied connectivity among these modules.

We used benchmarks from the multimedia and DSP domains for our experiments. The list of the benchmarks is shown in Table IV. The benchmarks are compiled using the EXPRESS compiler. We collected the statistics information using the SIMPRESS cycle-accurate simulator, which models both the TIC6211 processor and the memory subsystem.

We used a greedy algorithm to modify the ADL description of the memory architecture for each exploration run. The compiler and simulator automatically extract the necessary parameters (e.g., cache parameters, connectivity, and so on) from the ADL description. Each memory parameter described in ADL is modified in powers of 2. For each module we used certain heuristics for size limitations. For example, when certain program or data cache returns 98% hit ratio for a set of application programs we do not increase its size any more. We obtain a new memory configuration by adding a new module, or by changing a parameter of an existing module, or by modifying the connectivity. However, as we explained earlier, any of the existing exploration algorithms and pruning techniques can be used to generate the ADL description during DSE.

7.2 Estimation Models

There are different kinds of estimation models available in the literature for area and energy computations. Each of these models are specific to certain types of architectures. While any micro-architectural estimation models can be used in our framework, we use area models from Mulder et al. [1991] and energy models from Wattch [Brooks et al. 2000]. These models are adapted to enable estimation of wide variety of memory configurations available in DSP, RISC, VLIW, and Superscalar architectures. The estimation models for performance, area, and energy are described below.

7.2.1 *Performance Computation.* The performance of a particular memory configuration for a given application program is the number of clock cycles it takes to execute the application in the cycle-accurate structural simulator SIMPRESS [Khare et al. 1999]. We divide this cycle count by 2000 to show both energy and performance plots in the same figure.

7.2.2 *Area Computation.* We have used the area model of Mulder et al. [1991] to compute the silicon area occupied by each memory configuration. The unit for the area model is a technology independent notion of a register-bit equivalent or *rbe*. The advantage of this is the relatively straightforward relation between area and size, facilitating interpretation of area figures. One *rbe* equals the area of a bit storage cell.

We present here the area model for set-associative cache and SRAM that we have used during area computation of memory configurations. We use the area model for set-associative cache to compute area for stream buffer as well. The area model for other memory components can be found in Mulder et al. [1991]. The area equation for a static memory with memory array $size_w$ words each of $line_b$ bits long is

$$area_{sram} = 0.6(size_w + 6)(line_b + 6)\ rbe.$$

The area for a set-associative cache is a function of the storage capacity $size_b$, the degree of associativity '*assoc*', the line size $line_b$, and the size of a transfer-unit $transfer_b$. The area of a set-associative cache using static ($area_{sac}^{static}$) and dynamic cells ($area_{sac}^{dynamic}$) are given below using the number of transfer units in a line *tunits*, the total number of address tags '*tags*', and the total number of tag and status bits tsb_b. Here, γ equals 2 for a write-back cache and 1 for a write-through cache.

$$\begin{aligned}
area_{sac}^{static} &= 195 + 0.6 \times ovhd_1 \times size_b + 0.6 \times ovhd_2 \times tsbits\ rbe\\
area_{sac}^{dynamic} &= 195 + 0.3 \times ovhd_3 \times size_b + 0.3 \times ovhd_4 \times tsbits\ rbe\\
tunits &= \frac{line_b}{transfer_b}\\
tags &= \frac{size_b}{line_b}\\
tsbits &= tsb_b \times tags = \left(1 + \gamma \times tunits + log_2 \frac{2^{30} \times assoc}{size_b}\right) \times tags\\
ovhd_1 &= 1 + \frac{6 \times assoc}{tags} + \frac{6}{line_b \times assoc}\\
ovhd_2 &= 1 + \frac{12 \times assoc}{tags} + \frac{6}{tsb_b \times assoc}\\
ovhd_3 &= 1 + \frac{6 \times assoc}{tags} + \frac{12}{line_b \times assoc}\\
ovhd_4 &= 1 + \frac{12 \times assoc}{tags} + \frac{12}{tsb_b \times assoc}.
\end{aligned}$$

7.2.3 *Energy Computation.* We use the power models described in Wattch [Brooks et al. 2000] for computation of energy dissipation in array structures in memory configurations. We briefly explain the power models proposed in Wattch. In CMOS-based logic circuits, dynamic power consumption P_d is the main source of power consumption, and is defined as: $P_d = CV_{dd}^2 af$. Here, C is the load capacitance, V_{dd} is the supply voltage, and f is the clock frequency. The activity factor, a, is a fraction between 0 and 1 indicating how often clock ticks lead to switching activity on average. C is calculated based on the circuit and the transistor sizings as described below. V_{dd} and f depend on the assumed process technology. The technology parameters for 0.35μ process are used from Palacharla et al. [1997].

The array structure power model is parameterized based on the number of rows (entries), columns (width of each entry), and the number of read/write ports. These parameters affect the size, number of decoders, number of wordlines, and number of bitlines. In addition, these parameters are used to estimate the length of the predecode wires as well as the lengths of the wordlines and bitlines that determine the capacitive loading on the lines.

The capacitances are modeled in Wattch using assumptions that are similar to those made by Wilton and Jouppi [1994] and Palacharla et al. [1997] in which the authors performed delay analysis on many units. In both of the above works, the authors reduced the units into stages and formed RC circuits for each stage. This allowed them to estimate the delay for each stage, and by summing these, the delay for the entire unit.

Similar steps are performed for the power analysis in Wattch with two key differences. First, they are only interested in the capacitance of each stage, rather than both R and C. Second, in Wattch the power consumption of *all* paths are analyzed and summed together. This is in contrast with the delay analysis approach in Wilton and Jouppi [1994], where the expected critical path is of interest. The analytical model for the capacitance estimation for wordline (WL) and bitline (BL) are given below.

$$\begin{aligned} WL\ Capacitance &= C_{diff}(WLDriver) + C_{gate}(CellAccess) \times NumBitLines \\ &\quad + C_{metal} \times WLLength \\ BL\ Capacitance &= C_{diff}(PreCharge) + C_{diff}(CellAccess) \times NumWLines \\ &\quad + C_{metal} \times BLLength. \end{aligned}$$

Figure 5 shows a schematic of the wordlines and bitlines in the array structure. For more details on computation of power consumption in array structures, refer to Brooks et al. [2000]

7.3 Results

Some of the configurations we experimented with are presented in Table V. Each row of the table corresponds to a memory configuration. The second column presents the area of the memory configurations. The remaining entries in the table represent the size of the memory module (e.g., the size of L1 in configuration 1 is 256 bytes) and the cache/stream buffer organizations: $num\_lines \times line\_size \times num\_ways \times word\_size$. The LRU cache replacement policy is used. The latency is defined in number of processor cycles. Note that, for stream buffer the num_ways represents the number of FIFO queues present in it. The first configuration contains an L1 instruction cache (256 bytes), L1 data cache (256 bytes), and a unified L2 cache (8K bytes). All the configurations contain the same off-chip DRAM module with a latency of 50 cycles. The cache sizes are decided based on the application programs. We used benchmarks from the multimedia and DSP domains for our experiments. These benchmarks are

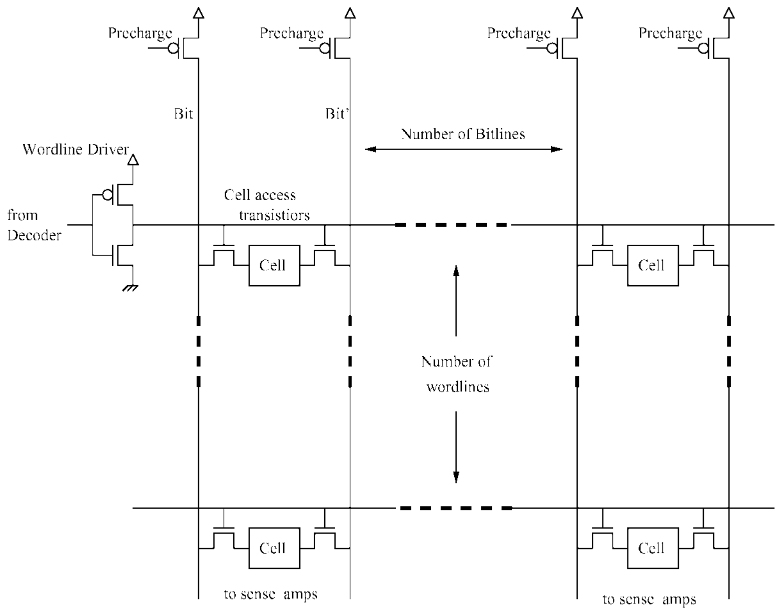


Fig. 5. Schematic of wordlines and bitlines in array structures.

Table V. The Memory Subsystem Configurations

Cfg	Area (rbe)	L1 ICache (latency = 1)	L1 DCache (latency = 1)	L2 Cache (latency = 5)	SRAM (lat = 1)	Str. Buffer (lat = 5)
1	54,567	256 B (8 × 2 × 4 × 4)	256 B (8 × 2 × 4 × 4)	8K (256 × 8 × 1 × 4)	–	–
2	61,335	256 B (8 × 2 × 4 × 4)	256 B (8 × 2 × 4 × 4)	4K (64 × 2 × 8 × 4)	4K	–
3	60,449	256 B (8 × 2 × 2 × 4)	256 B (8 × 2 × 4 × 4)	8K (64 × 4 × 8 × 4)	–	–
4	66,394	256 B (8 × 2 × 4 × 4)	256 B (8 × 2 × 4 × 4)	8K (64 × 4 × 8 × 4)	–	8 × 2 × 8 × 4
5	125,466	256 B (8 × 2 × 4 × 4)	256 B (8 × 2 × 4 × 4)	2K (16 × 4 × 8 × 4)	16K	8 × 2 × 8 × 4
6	51,169	128 B (8 × 2 × 2 × 4)	128 B (8 × 2 × 2 × 4)	8K (256 × 8 × 1 × 4)	–	–
7	52,868	128 B (8 × 2 × 2 × 4)	256 B (8 × 2 × 4 × 4)	8K (256 × 8 × 1 × 4)	–	–
8	58,198	128 B (8 × 2 × 2 × 4)	256 B (8 × 4 × 2 × 4)	8K (64 × 4 × 8 × 4)	–	–
9	52,057	128 B (8 × 2 × 2 × 4)	256 B (16 × 2 × 2 × 4)	8K (256 × 8 × 1 × 4)	–	–
10	52,868	256 B (8 × 2 × 4 × 4)	128 B (8 × 2 × 2 × 4)	8K (256 × 8 × 1 × 4)	–	–
11	33,099	256 B (8 × 4 × 2 × 4)	256 B (8 × 4 × 2 × 4)	4K (64 × 4 × 4 × 4)	–	–
12	31,698	256 B (16 × 2 × 2 × 4)	256 B (16 × 2 × 2 × 4)	4K (256 × 4 × 1 × 4)	–	–
13	33,469	256 B (16 × 2 × 2 × 4)	512 B (32 × 2 × 2 × 4)	4K (256 × 4 × 1 × 4)	–	–
14	53,847	512 B (16 × 4 × 2 × 4)	128 B (8 × 2 × 2 × 4)	8K (256 × 8 × 1 × 4)	–	–
15	33,488	512 B (16 × 4 × 2 × 4)	256 B (16 × 2 × 2 × 4)	4K (256 × 4 × 1 × 4)	–	–
16	35,259	512 B (16 × 4 × 2 × 4)	512 B (32 × 2 × 2 × 4)	4K (256 × 4 × 1 × 4)	–	–
17	58,100	512 B (8 × 8 × 2 × 4)	512 B (8 × 8 × 2 × 4)	8K (256 × 8 × 1 × 4)	–	–
18	36,066	512 B (8 × 8 × 2 × 4)	512 B (16 × 4 × 2 × 4)	4K (256 × 4 × 1 × 4)	–	–
19	59,156	512 B (8 × 4 × 4 × 4)	512 B (8 × 4 × 4 × 4)	8K (256 × 8 × 1 × 4)	–	–
20	55,182	256 B (16 × 2 × 2 × 4)	256 B (16 × 2 × 2 × 4)	4K (256 × 4 × 1 × 4)	4K	–
21	53,406	128 B (8 × 2 × 2 × 4)	128 B (8 × 2 × 2 × 4)	4K (256 × 4 × 1 × 4)	4K	–
22	76,753	128 B (8 × 2 × 2 × 4)	128 B (8 × 2 × 2 × 4)	4K (256 × 4 × 1 × 4)	8K	–
23	36,227	128 B (8 × 2 × 2 × 4)	256 B (16 × 2 × 2 × 4)	4K (256 × 4 × 1 × 4)	–	8 × 8 × 2 × 4
24	33,909	128 B (8 × 2 × 2 × 4)	256 B (16 × 2 × 2 × 4)	8K (256 × 4 × 1 × 4)	–	8 × 4 × 2 × 4
25	106,722	128 B (8 × 2 × 2 × 4)	256 B (16 × 2 × 2 × 4)	8K (64 × 8 × 4 × 4)	8K	8 × 8 × 2 × 4
26	106,722	128 B (8 × 2 × 2 × 4)	256 B (16 × 2 × 2 × 4)	8K (64 × 8 × 4 × 4)	8K	8 × 8 × 2 × 4
27	85,146	128 B (8 × 2 × 2 × 4)	512 B (32 × 2 × 2 × 4)	8K (64 × 8 × 4 × 4)	4K	8 × 8 × 2 × 4

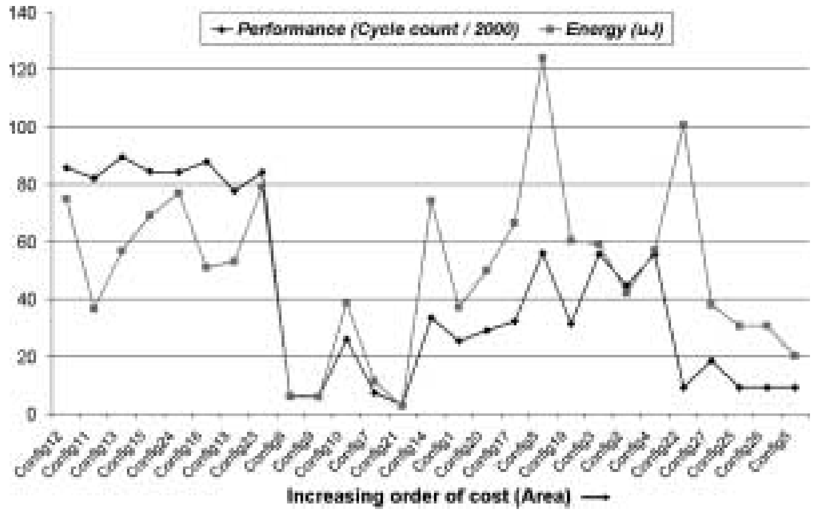


Fig. 6. Memory exploration results for GSR.

small/medium size kernels. Therefore, the cache sizes are smaller than typical sizes used in processor architectures.

Here we analyze a subset of the experiments we ran with the goal of evaluating different memory configurations for area, energy, and performance. Figure 6 shows the exploration result for the *GSR* benchmark. The X-axis represents the memory configurations in the increasing order of cost in terms of area. The Y-axis represents values for both performance and energy. The performance value is normalized by dividing cycle count by 2000. The energy value is given in μJ. Although the cost for memory configurations 6 and 9 are much lower than the cost of configuration 5, the former (6 and 9) configurations deliver better results in terms of energy and performance. Configuration 21 consumes lower energy and delivers better performance than configuration 6. However, the former is worse than the latter in terms of area. Depending on the priority among area, energy and performance, one of the configurations can be selected.

When area consideration is not very important we can view the pareto-optimal configurations from energy–performance trade-offs. Figure 7 shows the energy–performance trade-off for *Compress* benchmark. It is interesting to note that a set a memory configurations (with varied parameters, modules, and connectivity) deliver similar performance results for *Compress* benchmark. As we can see that there are three distinct performance zones. The first zone has performance values between 5 and 10. This zone consists of memory configurations 2, 5, 20, 21, 22, 25, 26, and 27. The power values are different due to the fact that each configuration has different parameters, modules, connectivity, and area. However, the performance is almost similar since the data fits in SRAM of size 2K for these configurations. Similarly, the second zone (configurations 1, 6, 7, 9, 10, 14, 17, and 19) has performance values between 15 and 20 with very different power values. The performance is almost same for these configurations because the L2 cache size of 8K or larger has very high hit ratio

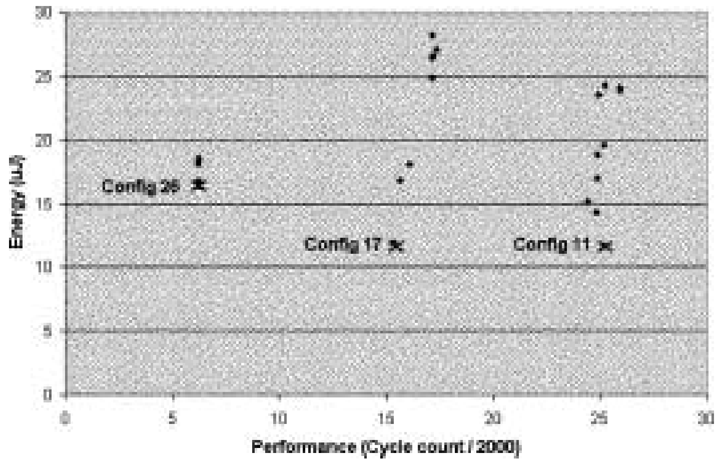


Fig. 7. Energy–performance trade-off for Compress.

and as a result for all these memory configurations L2 dominates and L2 to DRAM access remains almost constant.

Similarly, the third zone (configurations 3, 4, 8, 11, 12, 13, 15, 16, 18, 23, and 24) has almost same performance with different power values. This is due to the fact that each of these configurations has L2 line size of 4 that dominates over other parameters for these configurations. This line size is the reason why configurations in third zone are worse than the configurations in the second zone. Depending on the priority among cost, energy, and performance one of the three configurations (Config 11, 17, 26) can be chosen. The same phenomenon can be observed in the benchmarks *FirstSum*, *FirstDiff*, and *FirstMin* [Mishra et al. 2002]. The benchmark *InnerProd* has four such zones whereas the benchmark *Tridiag* has five such performance zones [Mishra et al. 2002]. The pareto-optimal configurations are shown using symbol *X*, and the corresponding memory configuration is mentioned in the figure.

However, for some set of benchmarks the energy–performance trade-off points are scattered in the design space and thus the pareto-optimal configurations are of interest. Figure 8 shows the energy–performance trade-off for the benchmark MatMult. It has only one pareto-optimal point, that is, configuration 5. However, the Laplace benchmark (Figure 9) has two pareto-optimal points. The configuration 5 delivers better performance than configuration 17 but consumes more energy and has larger area requirement. Depending on the priority among area, energy, and performance, one of the two configurations can be selected. The energy–performance trade-off results for the remaining benchmarks are shown in Mishra et al. [2002].

Thus, using our Memory-Aware ADL-based DSE approach, we obtained design points with varying cost, energy, 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, memory-aware compiler and simulator should be automatically

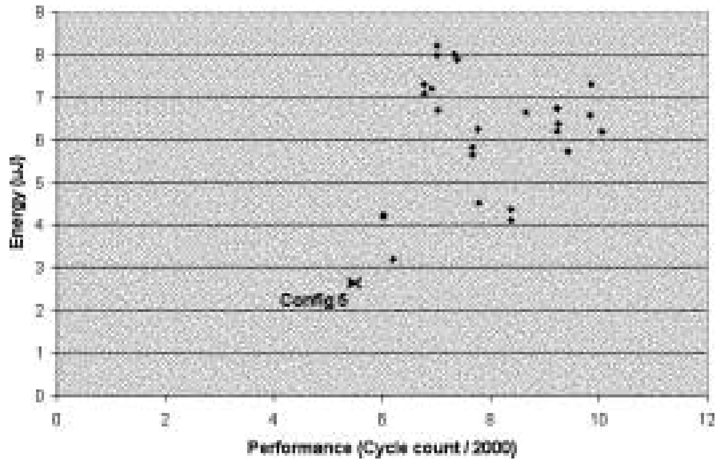


Fig. 8. Energy–performance trade-off for MatMult.

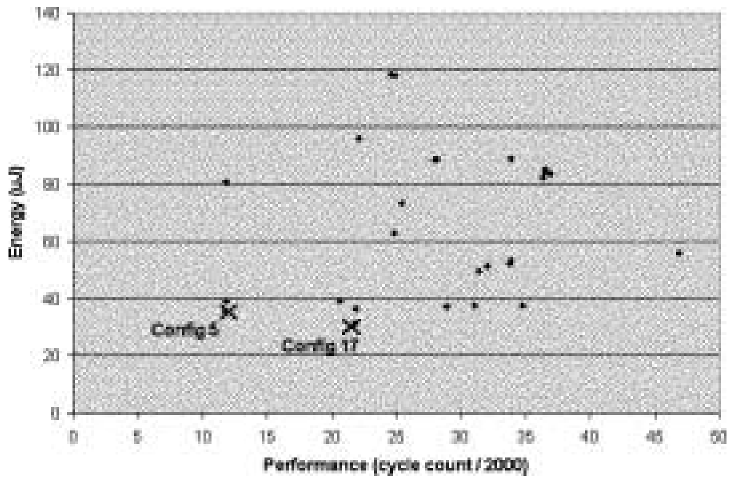


Fig. 9. Energy–performance trade-off for Laplace.

generated, and the applications have to be executed on the configured processor–memory system, as we demonstrated in this section.

8. CONCLUSIONS

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, and so on). On the other hand, new memory modules and organizations provide a set of features which exploit specific application needs (e.g., caches, stream buffers, page-mode, burst-mode, and 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, and performance). Performing such processor–memory coexploration requires the capability to capture the memory subsystem, generate memory aware software toolkit, and perform a compiler-in-the-loop exploration/evaluation.

In this paper we presented an ADL that captures the memory subsystem explicitly. The ADL is used to drive the generation of a memory-aware software toolkit including compiler and simulator, and also facilitate the exploration of various memory configurations. We obtained design points with varying cost, energy, and performance attributes using ADL-driven processor–memory coexploration.

Our ongoing work targets the use of this ADL-driven 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.

ACKNOWLEDGMENTS

This work was partially supported by NSF grants CCR-0203813 and CCR-0205712. We would like to thank Prof. Alex Nicolau, Dr. Peter Grun and Ashok Halambi for their insightful comments. We would like to acknowledge the members of the ACES laboratory for their inputs.

REFERENCES

ARC. ARC cores. http://www.arccores.com.

Ascia, G., Catania, V., and Palesi, M. 2001. Parameterized system design based on genetic algorithms. In *Proceedings of Hardware/Software Codesign (CODES)*. 177–182.

Axys. Axys Design Automation. http://www.axysdesign.com.

Brooks, D., Tiwari, V., and Martonosi, M. 2000. Wattch: A framework for architectural-level power analysis and optimizations. In *Proceedings of International Symposium on Computer Architecture (ISCA)*, 83–94.

Catthoor, F., Wuytack, S., Greef, E., Balasa, F., Nachtergaele, L., and Vandecappelle, A. 1998. *Custom Memory Management Methodology*. Kluwer Academic, Norwell, MA.

Chang, P., Mahlke, S., Chen, W., Water, N., and Hwu, W. 1991. IMPACT: An architectural framework for multiple-instruction-issue processors. In *Proceedings of International Symposium on Computer Architecture (ISCA)*, 266–275.

Fornaciari, W., Sciuto, D., Silvano, C., and Zaccaria, V. 2001. A design framework to efficiently explore energy delay tradeoffs. In *Proceedings of International Symposium on Hardware/Software Codesign (CODES)*, 260–265.

Freericks, M. 1993. The nML machine description formalism. Tech. Rep. TR SM-IMP/DIST/08, TU Berlin CS Dept.

Givargis, T., Wahid, F., and Henkel, J. 2001. System-level exploration for Pareto-optimal configurations in parameterized system-on-a-chip. In *Proceedings of International Conference on Computer-Aided Design (ICCAD)*, 25–30.

Green Hills Software Inc. 2003. http://www.ghs.com. Green Hills Software Inc.

Grun, P., Dutt, N., and Nicolau, A. 2000a. Memory aware compilation through accurate timing extraction. In *Proceedings of Design Automation Conference (DAC)*, 316–321.

Grun, P., Dutt, N., and Nicolau, A. 2000b. MIST: An algorithm for memory miss traffic management. In *Proceedings of International Conference on Computer-Aided Design (ICCAD)*, 431–437.

Grun, P., Dutt, N., and Nicolau, A. 2001. APEX: Access pattern based memory architecture customization. In *Proceedings of International Symposium on System Synthesis (ISSS)*, 25–32.

GRUN, P., DUTT, N., AND NICOLAU, A. 2002. Memory system connectivity exploration. In *Proceedings of Design Automation and Test in Europe (DATE)*, 894–901.

GRUN, P., HALAMBI, A., DUTT, N., AND NICOLAU, A. 1999. RTGEN: An algorithm for automatic generation of reservation tables from architectural descriptions. In *Proceedings of International Symposium on System Synthesis (ISSS)*, 44–50.

HADJIYIANNIS, G., HANONO, S., AND DEVADAS, S. 1997. ISDL: An instruction set description language for retargetability. In *Proceedings of Design Automation Conference (DAC)*, 299–302.

HALAMBI, A., GRUN, P., GANESH, V., KHARE, A., DUTT, N., AND NICOLAU, A. 1999. EXPRESSION: A language for architecture exploration through compiler/simulator retargetability. In *Proceedings of Design Automation and Test in Europe (DATE)*, 485–490.

IBM0316409C. IBM Microelectronics, data sheets for synchronous DRAM IBM0316409C. http://www.chips.ibm.com/products/memory/08J3348.

INOUE, A., TOMIYAMA, H., EKO, F., KANBARA, H., AND YASUURA, H. 1998. A programming language for processor-based embedded systems. In *Proceedings of Asia Pacific Conference on Hardware Description Languages (APCHDL)*, 89–94.

JOUPPI, N. 1990. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *Proceedings of International Symposium on Computer Architecture (ISCA)*, 364–373.

KHARE, A., SAVOIU, N., HALAMBI, A., GRUN, P., DUTT, N., AND NICOLAU, A. 1999. V-SAT: A visual specification and analysis tool for system-on-chip exploration. In *Proceedings of EUROMICRO Conference*, 1196–1203.

LEE, C., KIN, J., POTKONJAK, M., AND MANGIONE-SMITH, W. 1998. Media architecture: General purpose vs. multiple application-specific programmable processor. In *Proceedings of Design Automation Conference (DAC)*, 321–326.

LEUPERS, R. AND MARWEDEL, P. 1997. Retargetable generation of code selectors from HDL processor models. In *Proceedings of European Design and Test Conference (EDTC)*, 140–144.

LIEVERSE, P., WOLF, P., DEPRETTERE, E., AND VISSERS, K. 1999. A methodology for architecture exploration of heterogeneous signal processing systems. In *Proceedings of Signal Processing Systems (SiPS)*, 181–190.

MISHRA, P., GRUN, P., DUTT, N., AND NICOLAU, A. 2000. Memory subsystem description in EXPRESSION. Tech. Rep. UCI-ICS 00-31, University of California, Irvine.

MISHRA, P., GRUN, P., DUTT, N., AND NICOLAU, A. 2001. Processor-memory co-exploration driven by an architectural description language. In *Proceedings of International Conference on VLSI Design*, 70–75.

MISHRA, P., MAMIDIPAKA, M., AND DUTT, N. 2002. A framework for memory subsystem exploration. Tech. Rep. CECS 02-19, University of California, Irvine.

MULDER, J. M., QUACH, N. T., AND FLYNN, M. J. 1991. An area model for on-chip memories and its application. *IEEE Journal of Solid State Circuits* SC-26, 1, 98–105.

PALACHARLA, S., JOUPPI, N., AND SMITH, J. 1997. Complexity-effective superscalar processors. In *Proceedings of International Symposium on Computer Architecture (ISCA)*, 206–218.

PANDA, P., DUTT, N., AND NICOLAU, A. 1997. Architectural exploration and optimization of local memory in embedded systems. In *Proceedings of International Symposium on System Synthesis (ISSS)*, 90–97.

RAJESH, V. AND MOONA, R. 1999. Processor modeling for hardware software codesign. In *Proceedings of International Conference on VLSI Design*, 132–137.

SHIUE, W. AND CHAKRABARTI, C. 1999. Memory exploration for low power embedded systems. In *Proceedings of Design Automation Conference (DAC)*, 140–145.

SIA. 1998. *National Technology Roadmap for Semiconductors: Technology Needs*. Semiconductor Industry Association.

SISKA, C. 1998. A processor description language supporting retargetable multi-pipeline DSP program development tools. In *Proceedings of International Symposium on System Synthesis (ISSS)*, 31–36.

SLOCK, P., WUYTACK, S., CATTHOOR, F., AND JONG, G. 1997. Fast and extensive system-level memory exploration for ATM applications. In *Proceedings of International Symposium on System Synthesis (ISSS)*, 74–81.

TARGET. http://www.retarget.com. Target Compiler Technologies.

TENSILICA. http://www.tensilica.com. Tensilica Inc.

Texas Instruments. 1998. *TMS320C6201 CPU and Instruction Set Reference Guide*. Texas Instruments.

TRIMARAN. 1997. *The MDES User Manual*. http://www.trimaran.org.

WILTON, S. AND JOUPPI, N. 1994. An Enhanced Access and Cycle Time Model for On-Chip Caches. Tech. Rep. 93/5, DEC Western Research Laboratory.

ZIVOJNOVIC, V., PEES, S., AND MEYR, H. 1996. LISA—Machine description language and generic machine model for HW/SW co-design. In *IEEE Workshop on VLSI Signal Processing*, 127–136.

Received August 2002; revised June 2003; July 2003; accepted August 2003