A Framework for Memory Subsystem Exploration

Prabhat Mishra	Mahesh Mamidipaka	Nikil Dutt
pmishra@cecs.uci.edu	maheshmn@cecs.uci.edu	dutt@cecs.uci.edu

Architectures and Compilers for Embedded Systems (ACES)
Center for Embedded Computer Systems, University of California, Irvine, CA, USA

CECS Technical Report #02-19
Center for Embedded Computer Systems
University of California, Irvine, CA 92697, USA

May 24, 2002

Abstract

Memory represents a major bottleneck in modern embedded systems in terms of cost, power and performance. Traditionally, memory organizations for programmable 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 the application. However, such a processor-memory co-exploration 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, and perform exploration of the memory architecture to meet diverse requirements: low power, better performance, smaller die size etc. We present a set of experiments using our Memory-Aware Architectural Description Language to drive the exploration of the memory subsystem for the TI C6211 processor architecture, demonstrating a range of cost, performance, and energy attributes.

Contents

1 Introduction **4**

2 Related Work **5**

3 Our Approach **6**

4 Motivating Example **6**

5 The Memory Subsystem Description in EXPRESSION **8**

6 Example Memory Architecture **9**

7 Experiments **11**

7.1 Experimental Setup 12

7.2 Estimation Models 13

7.2.1 Performance Computation 13

7.2.2 Area Computation 13

7.2.3 Energy Computation 14

7.3 Results 15

8 Summary **19**

9 Acknowledgments **20**

A Appendix **22**

List of Figures

1	The Flow in our approach	7
2	A Motivating Example	8
3	Sample Memory Architecture for TIC6211	11
4	Schematic of wordlines and bitlines in array structures	15
5	Memory Exploration Results for GSR	17
6	Energy Performance Tradeoff for Compress	18
7	Energy Performance Tradeoff for Laplace	18
8	Energy Performance Tradeoff for MatMult	19
9	Energy Performance Tradeoff for 1dpartpush	22
10	Energy Performance Tradeoff for 2dhydro	22
11	Energy Performance Tradeoff for Condcompute	23
12	Energy Performance Tradeoff for Diffpred	23
13	Energy Performance Tradeoff for Firstdiff	23
14	Energy Performance Tradeoff for Firstmin	24
15	Energy Performance Tradeoff for Firstsum	24
16	Energy Performance Tradeoff for GLRE	24
17	Energy Performance Tradeoff for GSR	25
18	Energy Performance Tradeoff for Hydro	25
19	Energy Performance Tradeoff for Hydrodynamics	25
20	Energy Performance Tradeoff for ICCG	26
21	Energy Performance Tradeoff for Innerprod	26
22	Energy Performance Tradeoff for integrate	26
23	Energy Performance Tradeoff for Intpred	27
24	Energy Performance Tradeoff for Linear	27
25	Energy Performance Tradeoff for Lineareqn	27
26	Energy Performance Tradeoff for Partpush	28
27	Energy Performance Tradeoff for Planc	29
28	Energy Performance Tradeoff for Recurrence	29
29	Energy Performance Tradeoff for Stateexcerpt	30
30	Energy Performance Tradeoff for Tridiag	30
31	Energy Performance Tradeoff for Wavelet	30

List of Tables

1	Benchmarks	12
2	The memory subsystem configurations	16

1 Introduction

Memory represents a major cost, power and performance bottleneck for a large class of embedded systems [29]. 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 [31] requires more aggressive use of memory configurations, *customized* for the specific target applications. To address this problem, on one hand recent advances in memory technology have generated a plethora of new and efficient memory modules (e.g., SDRAM, DDRAM, RAMBUS, etc.), exhibiting a heterogeneous set of features (e.g., page-mode, burst-mode, 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 (transparent to the processor) 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 co-exploration of the processor and the memory architecture, it is possible to better exploit the heterogeneity in the memory subsystem organizations, and better trade-off system attributes such as cost, performance, and power. However, such processor-memory co-exploration requires the capability to explicitly capture, exploit, and refine both the processor as well as the memory architecture.

Recent approaches on language-driven Design Space Exploration (DSE) ([1], [3], [9], [14], [16], [23], [30], [34], [35]), use 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.

The contribution of this paper is the explicit description of a customized, heterogeneous memory architecture in our EXPRESSION ADL [23], permitting co-exploration 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 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 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 illustrates memory architecture exploration using experiments on the TIC6211 processor, with varying memory configurations to explore design points for cost, power and performance attributes. Section 8 concludes the paper.

2 Related Work

We discuss related research in two categories. First, we survey recent approaches on Architecture Description Language (ADL) driven Design Space Exploration, and second, we discuss previous works on embedded system exploration.

An extensive body of recent research addresses ADL driven software toolkit generation and Design Space Exploration (DSE) for processor-based embedded systems, in both academia: ISDL [9], Valen-C [10], MIMOLA [14], LISA [15], nML [16], [30], and industry: ARC [1], Axys [2], RADL [32], Target [33], Tensilica [34], MDES [35].

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, stream/prefetch buffers), their complex memory features (e.g., page-mode, burst-mode accesses), their detailed timings, resource utilization, and the overall organization of the memory architecture (e.g., multiple cache hierarchies, partitioned memory spaces, direct-mapped memories, etc.)

Memory exploration for embedded systems has been addressed by Panda et al. [27]. The metric used for the system are data cache size and number of processor cycles. The method has been extended by Shiue et al. [26] to include energy consumption as one of the metric. Catthoor et al. [20] have presented a methodology for memory hierarchy and data reuse decision exploration. Grun et al. proposed techniques for early memory [18] and connectivity [19] architecture exploration. A system level performance analysis and design space exploration methodology (SPADE) is proposed by Lieverse et al. [8]. In this methodology application tuning is driven manually by the designer. Several design space exploration approaches use heuristics to prune the potentially large design space. Givargis et al. [11] 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) there by reducing the search space. Ascia et al. [13] proposed a technique to map the exploration problem to a genetic algorithm. Fornaciari et al. [12] use a sensitivity based technique in which the sensitivity of the 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. Also, the memory modules considered are traditional cache hierarchies and SRAMs. Our framework allows exploration of generic memory configurations consisting of any memory connectivity and modules chosen from memory IP library. This memory subsystem exploration is performed along with any processor structure 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 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 (assumed an implicitly defined memory architecture, e.g., a fixed cache hierarchy) 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. However, by describing the pipelining and parallelism available in recent memory organizations, there is tremendous opportunity for the compiler to generate performance improvements. Figure 1 shows our processor-memory co-exploration framework. Our previous work on reservation table [22] and operation timing generation [17] 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 [17]. Such aggressive optimizations in the presence of efficient memory access modes (e.g., page/burst modes) and cache hierarchies [21] are only possible due to the explicit representation of the detailed memory architecture.

The contribution of this paper is the memory subsystem description in EXPRESSION that enables the compiler to exploit the memory features along with processor details. We also present the memory simulator generation (shown shaded in Figure 1) that is integrated into the SIMPRESS [25] simulator, allowing for detailed feedback on the memory subsystem architecture and its match to the target applications.

4 Motivating Example

A typical efficient 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 IBM0316409C [37] Synchronous DRAM. This memory contains 2 banks, organized as arrays of 2048 rows x 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

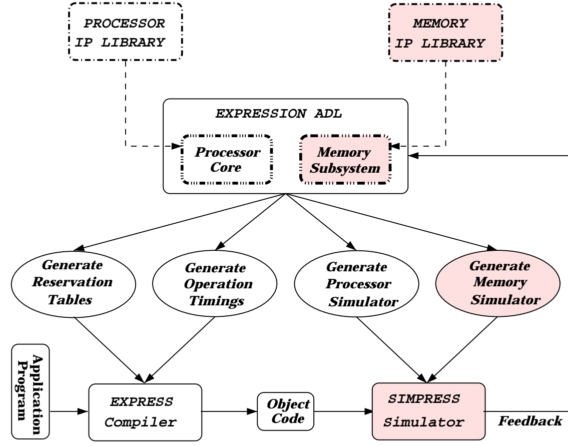

Figure 1. The Flow in our approach

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 which 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 more globally such parallelism, and better hide the latencies of the memory operations.

A sample code shown in (a) of Figure 2 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 access mode operations for a Synchronous DRAM are shown in (b) of Figure 2: the un-shaded 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 x (5 x 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 3 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

[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.

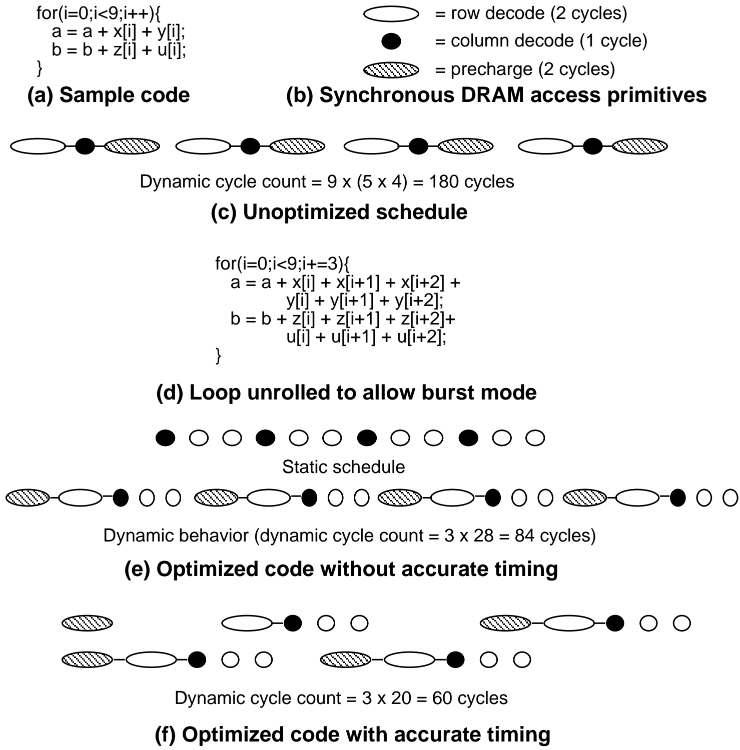

Figure 2. A Motivating Example

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 cycle-count in this case is 3 x 28 = 84 cycles.

Figure 2 (f) shows the effect of scheduling using accurate memory timing on code that has already been optimized for burst mode. Since 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 x 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.

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, store etc.), control instructions for the different memory components (such as the DMA), or explicit cache control instructions (e.g., cache freeze, prefetch, replace, refill etc.).

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).

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 varieties of architectures, including RISC, DSP, VLIW, and Superscalar. Further details on the memory subsystem description in EXPRESSION can be found in [28].

6 Example Memory Architecture

We illustrate our Memory-Aware Architectural Description Language (ADL) using the Texas Instruments TIC6211 VLIW DSP [36] processor that has several novel memory features. Figure 3 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. 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 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 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 [23]:

```
(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 [23]:

```
(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, but rather represent explicitly operations which are implicit in the processor and memory subsystem behavior.

The pipelining and parallelism between the cache operations are described in EXPRESSION through PIPELINE_PATHS [23]. Pipeline Paths represent the ordering between pipeline stages in the architecture (represented as bold arrows in Figure 3). 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:

```
(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 3.

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

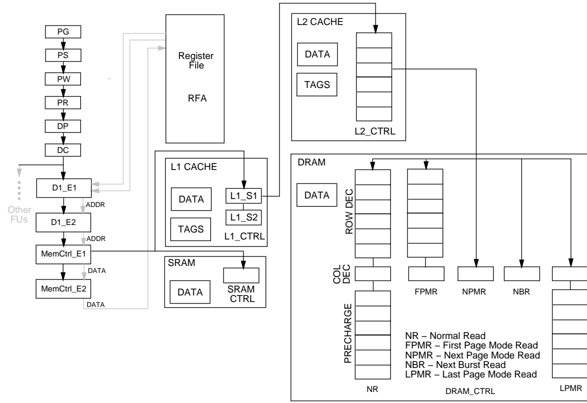

Figure 3. Sample Memory Architecture for TIC6211

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 [17]. 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_DEC is 6 cycles, of COL_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. The EXPRESSION description can be used to drive the generation of both a memory-aware compiler [17], [21], and cycle-accurate structural memory subsystem simulator, and thus enable Design Space Exploration and co-design of the memory and processor architecture. For more details on the memory subsystem description in EXPRESSION and automatic simulator generation, please refer to [28].

7 Experiments

As described earlier, we have already used this Memory-Aware Architectural Description Language (ADL) approach to generate a Memory-Aware Compiler [17] and manage the memory miss traffic [21], resulting in significantly improved performance. In this section we demonstrate further use of the memory subsystem specification to describe different memory configurations and perform design space exploration with the goal of evaluating different memory 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 [36] processor architecture, and varied the memory subsystem architecture. We generated a cycle-accurate simulator, and performed Design Space Exploration 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 [24] 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 1. 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.

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*matrix product
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 1. Benchmarks

We used a greedy algorithm to modify the ADL description of the memory architecture for each exploration run. The simulator extracts the necessary parameters (e.g., cache parameters, connectivity etc.) from the ADL description automatically for each exploration run. We modify each parameter value 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. However, as we explained earlier, any of the existing exploration algorithms and pruning techniques can be used to generate the ADL description during design space exploration.

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. We have used area models from Mulder et al. [4] and energy models from Wattch [5]. 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

We compute performance of a particular memory configuration for a given application program using the number of clock cycles it takes to execute the application in cycle-accurate structural simulator SIMPRESS [25] for that memory configuration. 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. [4] 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 [4]. The area equation for static memory with memory array $size_w$ words each of $line_b$ bits long is

$$area_{sram} = 0.6(size_w + 6)(line_b + 6)\ rbe \tag{1}$$

The area for 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.

$$area_{sac}^{static} = 195 + 0.6 \times ovhd_1 \times size_b + 0.6 \times ovhd_2 \times tsbitsrbe$$
$$area_{sac}^{dynamic} = 195 + 0.3 \times ovhd_3 \times size_b + 0.3 \times ovhd_4 \times tsbitsrbe$$
$$tunits = \frac{line_b}{transfer_b}$$
$$tags = \frac{size_b}{line_b}$$
$$tsbits_b = tsb_b \times tags = (1 + \gamma \times tunits + log_2 \frac{2^{30} \times assoc}{size_b}) \times tags$$
$$ovhd_1 = 1 + \frac{6 \times assoc}{tags} + \frac{6}{line_b \times assoc}$$
$$ovhd_2 = 1 + \frac{12 \times assoc}{tags} + \frac{6}{tsb_b \times assoc}$$
$$ovhd_3 = 1 + \frac{6 \times assoc}{tags} + \frac{12}{line_b \times assoc}$$
$$ovhd_4 = 1 + \frac{12 \times assoc}{tags} + \frac{12}{tsb_b \times assoc}$$

7.2.3 Energy Computation

We use the power models described in Wattch [5] for energy computation of array structures in memory configurations. We briefly explain the power models proposed in Wattch. In CMOS microprocessors, 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 0.35u process are used from [6].

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, the number of decoders, the number of wordlines, and the number of bitlines. In addition, these parameters are used to estimate the length of the pre-decode wires as well as the lengths of the wordlines and bitlines which determine the capacitive loading on the lines.

The capacitances are modeled in Wattch using assumptions that are similar to those made by [7] and [6] 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 [7], where the expected critical path is of interest. The analytical model

for the capacitance estimation for Wordline (WL) and Bitline (BL) are given below. Figure 4 shows a schematic of the wordlines and bitlines in the array structure.

$$WLCapacitance = C_{diff}(WLDriver) + C_{gate}(CellAccess) \times NumBitLines + C_{metal} \times WLLength$$
$$BLCapacitance = C_{diff}(PreCharge) + C_{diff}(CellAccess) \times NumWLines + C_{metal} \times BLLength$$

For more details on computation of power consumption refer to [5]

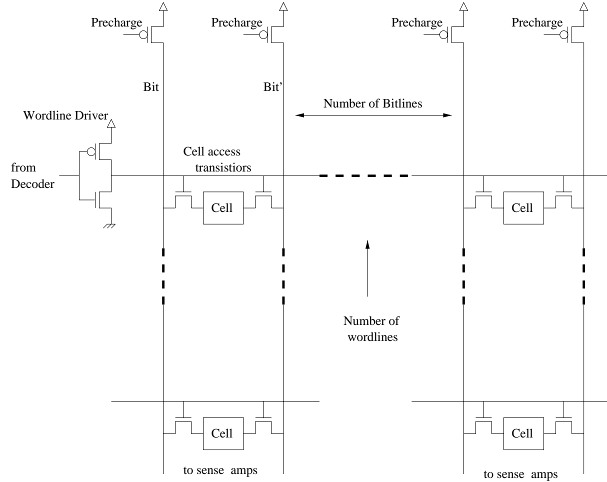

Figure 4. Schematic of wordlines and bitlines in array structures

7.3 Results

Some of the configurations we experimented with are presented in Table 2. The numbers in Table 2 represent: the size of the memory module (e.g., the size of L1 in configuration 1 is 256 bytes), the cache/stream buffer organizations: $num_lines \times line_size \times num_ways \times word_size$. The LRU replacement policy is used. The latency is defined in number of processor cycles. Note that for the stream buffer, num_ways represents the number of FIFO queues present.

The configurations in Table 2 are presented in increasing order of the cost in terms of area. We have used the area model of Mulder et al.[4] (as described in Section 7.2.2) to derive the area of each on-chip memory configuration. The unit for the area model is a technology independent notion of a register-bit equivalent or *rbe*. 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 150 cycles.

Here we analyze a subset of the experiments we ran with the goal of evaluating different memory configurations for cost power and performance. The power performance tradeoff results for the remaining benchmarks are shown in Section A. Figure 5 shows the exploration result for the *GSR* benchmark. The X-axis represents the memory configurations in the increasing order of cost. The

Config	Area (rbe)	L1 ICache (latency=1)	L1 DCache (latency=1)	L2 Cache (latency=5)	SRAM (latency=1)	Stream Buffer (latency=5)
1	54567	256B (8x2x4x4)	256B (8x2x4x4)	8K (256x8x1x4)	-	-
2	131376	256B (8x2x4x4)	256B (8x2x4x4)	4K (64x2x8x4)	4K	-
3	60449	256B (8x2x2x4)	256B (8x2x4x4)	8K (64x4x8x4)	-	-
4	66394	256B (8x2x4x4)	256B (8x2x4x4)	8K (64x4x8x4)	-	512B (8x2x8x4)
5	405632	256B (8x2x4x4)	256B (8x2x4x4)	2K (16x4x8x4)	16K	512B (8x2x8x4)
6	51169	128B (8x2x2x4)	128B (8x2x2x4)	8k (256x8x1x4)	-	-
7	52868	128B (8x2x2x4)	256B (8x2x4x4)	8k (256x8x1x4)	-	-
8	58198	128B (8x2x2x4)	256B (8x4x2x4)	8k (64x4x8x4)	-	-
9	52057	128B (8x2x2x4)	256B (16x2x2x4)	8k (256x8x1x4)	-	-
10	52868	256B (8x2x4x4)	128B (8x2x2x4)	8k (256x8x1x4)	-	-
11	33099	256B (8x4x2x4)	256B (8x4x2x4)	4k (64x4x4x4)	-	-
12	31698	256B (16x2x2x4)	256B (16x2x2x4)	4k (256x4x1x4)	-	-
13	33469	256B (16x2x2x4)	512B (32x2x2x4)	4k (256x4x1x4)	-	-
14	53847	512B (16x4x2x4)	128B (8x2x2x4)	8k (256x8x1x4)	-	-
15	33488	512B (16x4x2x4)	256B (16x2x2x4)	4k (256x4x1x4)	-	-
16	35259	512B (16x4x2x4)	512B (32x2x2x4)	4k (256x4x1x4)	-	-
17	58100	512B (8x8x2x4)	512B (8x8x2x4)	8k (256x8x1x4)	-	-
18	36066	512B (8x8x2x4)	512B (16x4x2x4)	8k (256x4x1x4)	-	-
19	59156	512B (8x4x4x4)	512B (8x4x4x4)	8k (256x8x1x4)	-	-
20	125223	256B (16x2x2x4)	256B (16x2x2x4)	4k (256x4x1x4)	4k	-
21	123447	128B (8x2x2x4)	128B (8x2x2x4)	4k (256x4x1x4)	4k	-
22	216836	128B (8x2x2x4)	128B (8x2x2x4)	4k (256x4x1x4)	8k	-
23	36227	128B (8x2x2x4)	256B (16x2x2x4)	4k (256x4x1x4)	-	512 (8x8x2x4)
24	33909	128B (8x2x2x4)	256B (16x2x2x4)	8k (256x4x1x4)	-	256 (8x4x2x4)
25	246805	128B (8x2x2x4)	256B (16x2x2x4)	8k (64x8x4x4)	8k	512 (8x8x2x4)
26	246805	128B (8x2x2x4)	256B (16x2x2x4)	8k (64x8x4x4)	8k	512 (8x8x2x4)
27	155188	128B (8x2x2x4)	512B (32x2x2x4)	8k (64x8x4x4)	4k	512 (8x8x2x4)

Table 2. The memory subsystem configurations

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 uJ. 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 in terms of area, energy and performance. The configuration 21 is better than configuration 6 in terms of energy and performance. However, the former is worse than the latter in terms of area. So depending on the priority among area, energy and performance, one of the configuration can be selected.

When area consideration is not very important we can view the pareto-optimal configurations from energy-performance trade-offs. Figure 6 shows the energy performance trade-offs 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 21, 20, 2, 27, 22, 25, 26 and 5. The power values are different due to the fact that each configuration has different parameters,

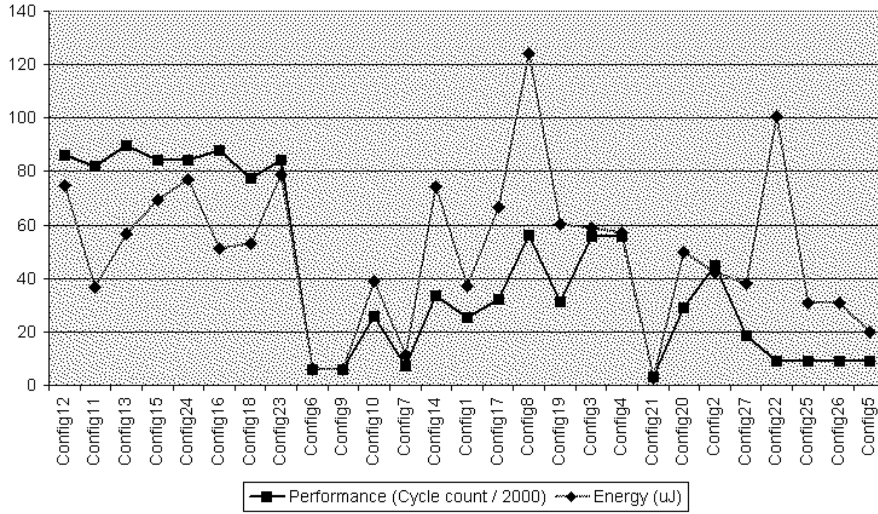

Figure 5. Memory Exploration Results for GSR

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 6, 9, 10, 7, 14, 1, 17, 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 and as a result for all these memory configurations L2 dominates and L2 to DRAM access remains almost constant. Similarly for the third zone (configurations 12, 11, 13, 15, 24, 16, 18, 23, 8, 3, 4) has almost same performance with different power values. This is due to the fact that each of these configuration has 4 as L2 line size that dominates over other parameters for these configurations. This line size is the reason why configurations in third zone is worse than configuration in second zone. The same phenomenon can be observed in the benchmarks *First-Sum*, *FirstDiff*, *FirstMin*. The benchmark *InnerProd* has four such zones whereas the benchmark *Tridiag* has five such performance zones. The pareto-optimal configurations are shown using symbol *X* and the corresponding memory configuration is mentioned in the figure. Depending of on the priority among cost, energy and performance one of three configurations (Config 11, 17, 26) can be chosen.

However, for some set of benchmarks the energy performance tradeoff points are scattered in the design space and thus the pareto-optimal configurations are of interest. Figure 8 shows the energy performance tradeoff for the benchmark MatMult. It has only one pareto-optimal point i.e., configuration 5. However, the benchmark Figure 7 has two pareto-optimal points. The configuration 5 consumes more energy and area than configuration 17. However, configuration 17 is better in terms of performance than configuration 5. Depending on the priority among area, energy and performance one of the two configurations can be selected.

Thus, using our Memory-Aware ADL-based Design Space Exploration 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

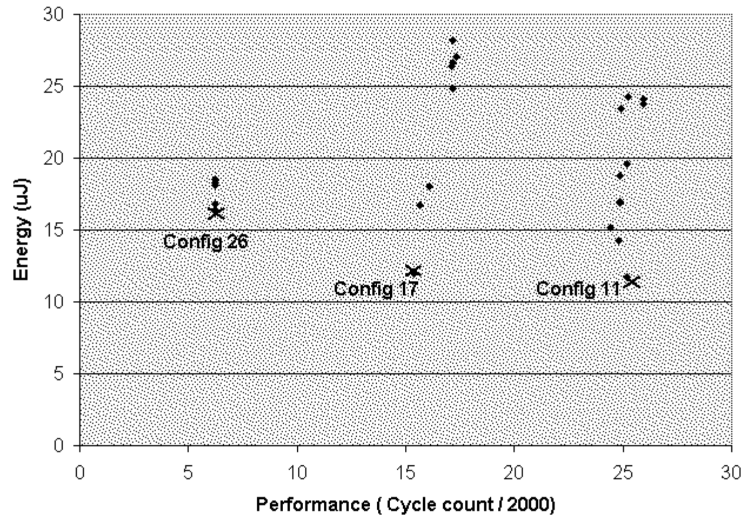

Figure 6. Energy Performance Tradeoff for Compress

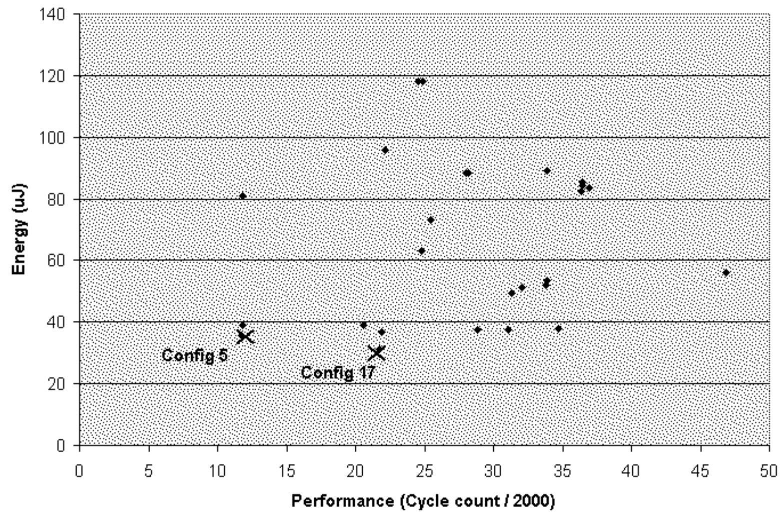

Figure 7. Energy Performance Tradeoff for Laplace

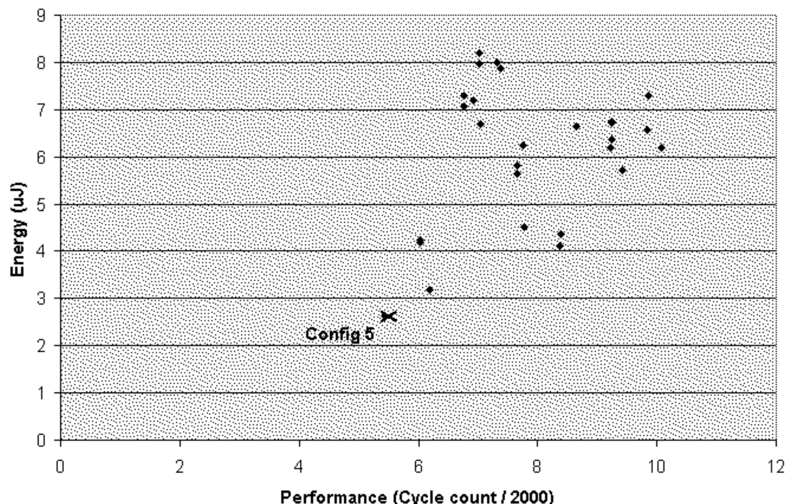

Figure 8. Energy Performance Tradeoff for MatMult

must be explicitly captured, and the applications have to be executed on the configured processor-memory system, as we demonstrated in this section.

8 Summary

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, and perform a compiler-in-the-loop exploration/evaluation.

In this paper 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.

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.

9 Acknowledgments

This work was partially supported by grants from DARPA (F33615-00-C-1632), Hitachi Ltd., and Motorola Inc. We would like to acknowledge Prof. Alex Nicolau, Dr. Peter Grun and Ashok Halambi for their contribution to the exploration work.

References

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

[2] Axys Design Automation. *http://www.axysdesign.com*.

[3] D. Lanneer et al. CHESS: Retargetable Code Generation for Embedded DSP Processors. *Code Generation for Embedded Processors.* Kluwer, 1997.

[4] J. Mulder et al. An Area Model for On-Chip Memories and its Application. *IEEE Journal of Solid-State Circuits*, Vol 26, No 2, pages 98–106, 1991.

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

[6] S. Palacharla et al. Complexity-Effective Superscalar Processors. *ISCA*, 1997.

[7] S. Wilton et al. An Enhanced Access and Cycle Time Model for On-Chip Caches. *WRL Research Report 93/5*, DEC Western Research Laboratory, 1994.

[8] P. Lieverse et al. A Methodology for Architecture Exploration of Heterogeneous Signal Processing Systems. *IEEE Workshop on Signal Processing Systems*, 1999.

[9] G. Hadjiyiannis et al. ISDL: An instruction set description language for retargetability. *DAC*, 1997.

[10] H. Yasuura et al. A programming language for processor based embedded systems. *APCHDL*, 1998.

[11] T. Givargis et al. System-Level Exploration for Pareto-Optimal Configurations in Parameterized System-on-a-Chip. *ICCAD*, 2001.

[12] W. Fornaciari et al. A Design Framework to Efficiently Explore Energy Delay Tradeoffs. *CODES*, 2001.

[13] G. Ascia et al. Parameterized System Design Based on Genetic Algorithms. *CODES*, 2001.

[14] R. Leupers et al. Retargetable generation of code selectors from HDL processor models. *EDTC*, 1997.

[15] V. Zivojnovic et al. LISA - machine description language and generic machine model for HW/SW co-design. *IEEE Workshop on VLSI Signal Processing*, 1996.

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

[17] P. Grun et al. Memory aware compilation through accurate timing extraction. *DAC*, 2000.

[18] P. Grun et al. APEX: Access Pattern based Memory Architecture Customization. *ISSS*, 2001.

[19] P. Grun et al. Memory Connectivity Architecture Exploration. *DATE*, 2001.

[20] F. Catthoor et al. Custom Memory Management Methodology. *Kluwer Academic Publishers*, 1998.

[21] P. Grun et al. Mist: An algorithm for memory miss traffic management. *ICCAD*, 2000.

[22] P. Grun et al. RTGEN: An algorithm for automatic generation of reservation tables from architectural descriptions. *ISSS*, 1999.

[23] A. Halambi et al. EXPRESSION: A language for architecture exploration through compiler/simulator retargetability. *DATE*, 1999.

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

[25] A. Khare et al. V-SAT: A visual specification and analysis tool for system-on-chip exploration. *EUROMICRO*, 1999.

[26] W. Shiue et al. Memory Exploration for Low Power Embedded Systems. *DAC*, 1999.

[27] P. Panda et al. Architectural Exploration and Optimization of Local Memory in Embedded Systems. *ISSS*, 1997.

[28] P. Mishra et al. Memory subsystem description in EXPRESSION. Technical Report UCI-ICS 00-31, University of California, Irvine, 2000.

[29] S. Przybylski. Sorting out the new DRAMs. *Hot Chips Tutorial*, Stanford, CA, 1997.

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

[31] Semiconductor Industry Association. *National technology roadmap for semiconductors: Technology needs*, 1998.

[32] C. Siska. A processor description language supporting retargetable multi-pipeline dsp program development tools. *Proc. ISSS*, 1998.

[33] Target Compiler Technologies. *http://www.retarget.com*.

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

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

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

[37] IBM Microelectronics, Data Sheets for Synchronous DRAM IBM0316409C. *www.chips.ibm.com/products/memory/08J3348/*.

A Appendix

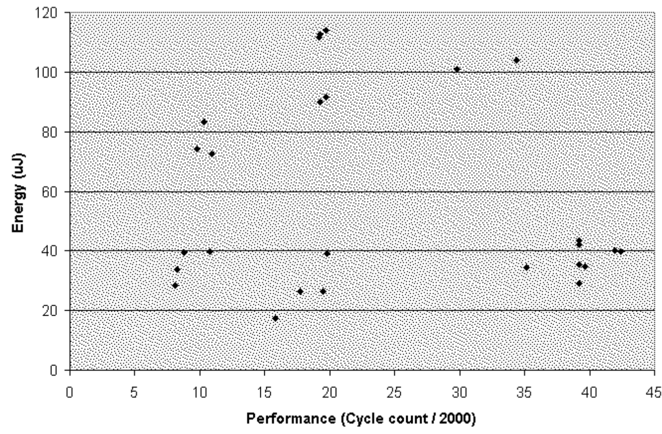

Figure 9. Energy Performance Tradeoff for 1dpartpush

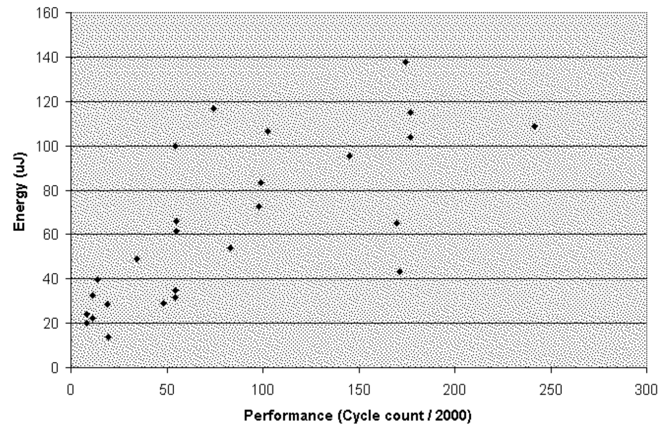

Figure 10. Energy Performance Tradeoff for 2dhydro

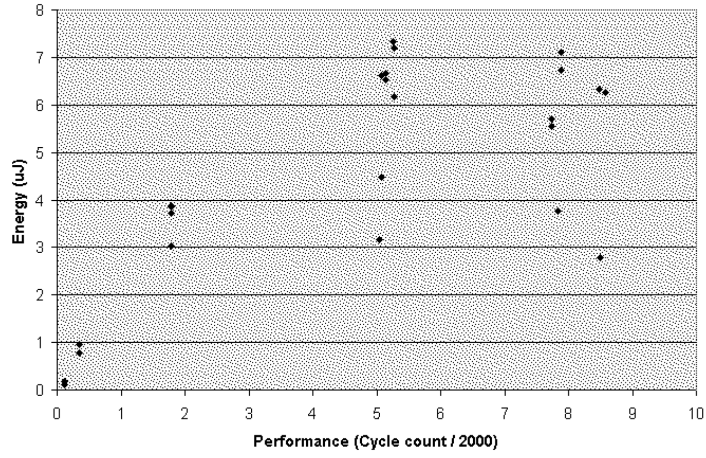

Figure 11. Energy Performance Tradeoff for Condcompute

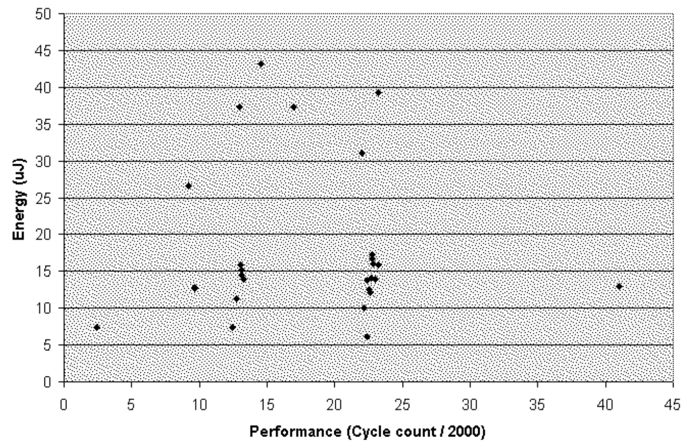

Figure 12. Energy Performance Tradeoff for Diffpred

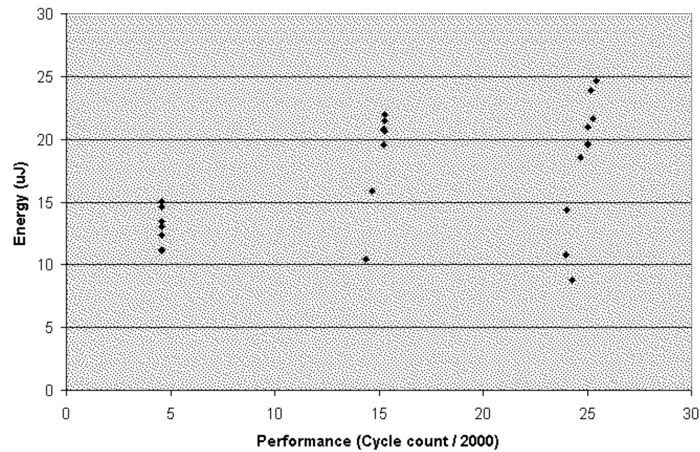

Figure 13. Energy Performance Tradeoff for Firstdiff

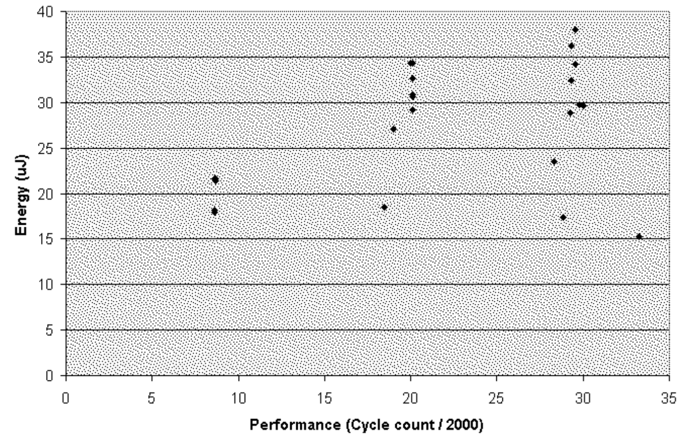

Figure 14. Energy Performance Tradeoff for Firstmin

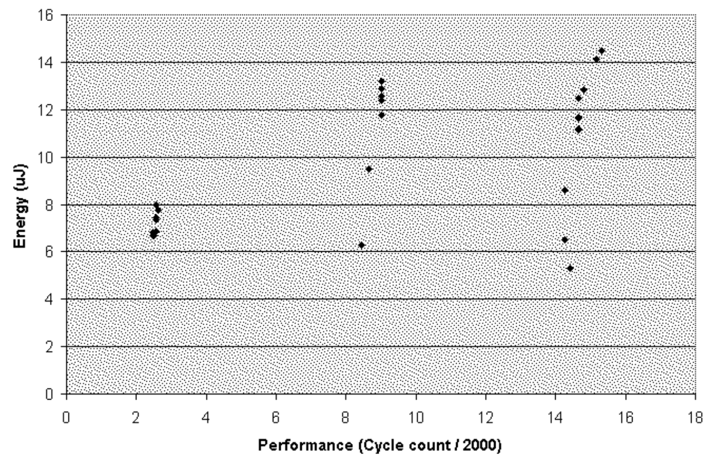

Figure 15. Energy Performance Tradeoff for Firstsum

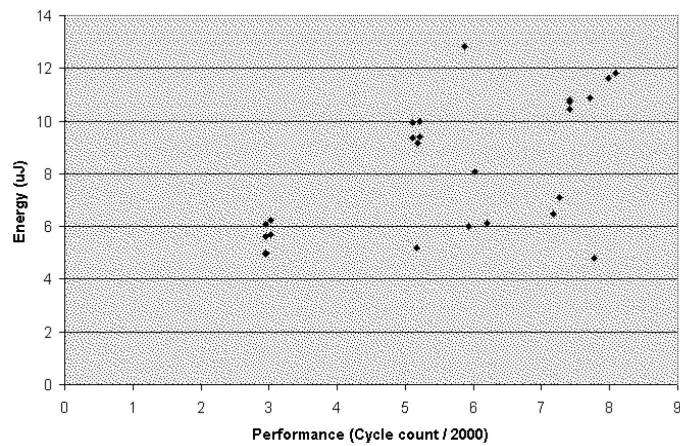

Figure 16. Energy Performance Tradeoff for GLRE

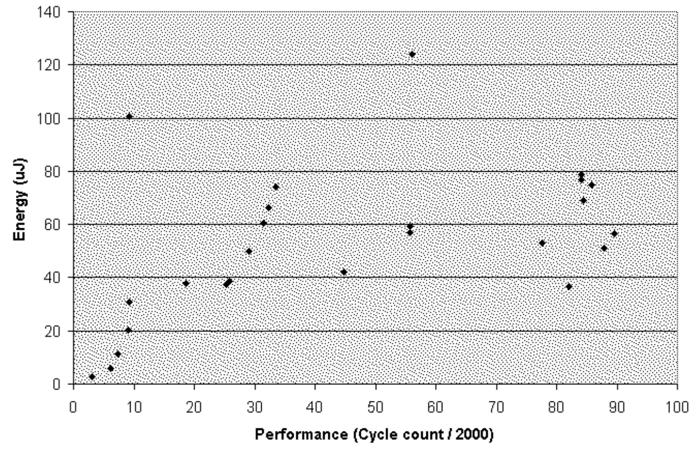

Figure 17. Energy Performance Tradeoff for GSR

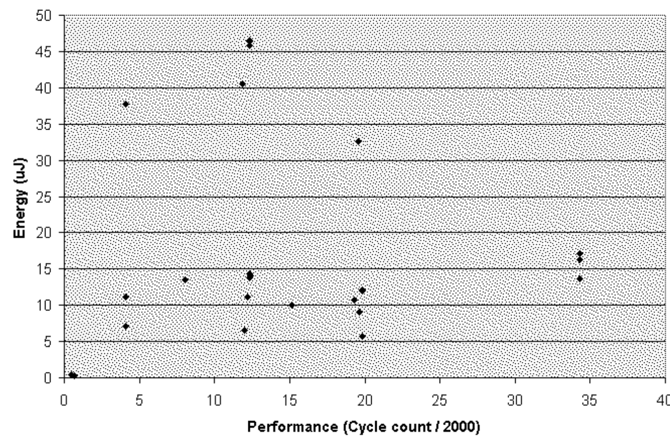

Figure 18. Energy Performance Tradeoff for Hydro

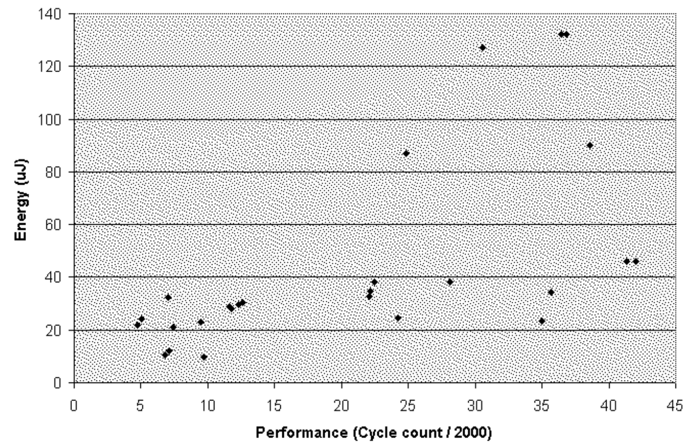

Figure 19. Energy Performance Tradeoff for Hydrodynamics

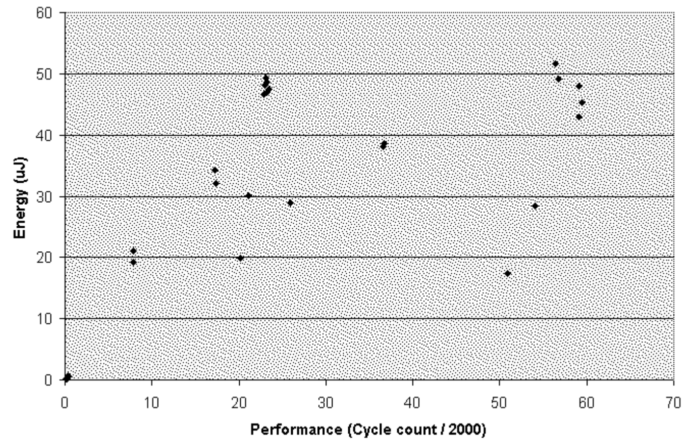

Figure 20. Energy Performance Tradeoff for ICCG

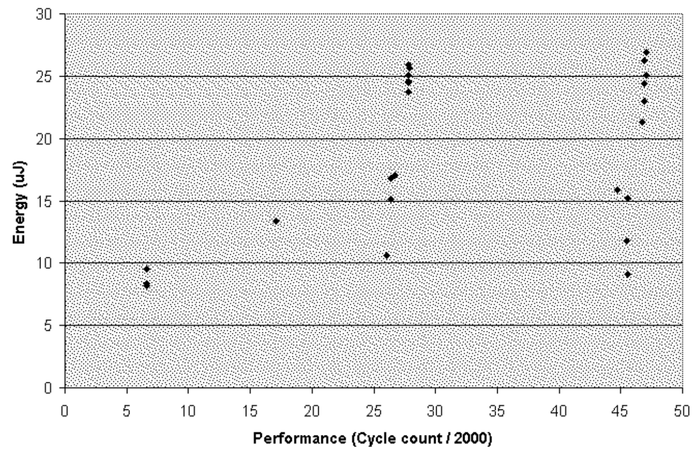

Figure 21. Energy Performance Tradeoff for Innerprod

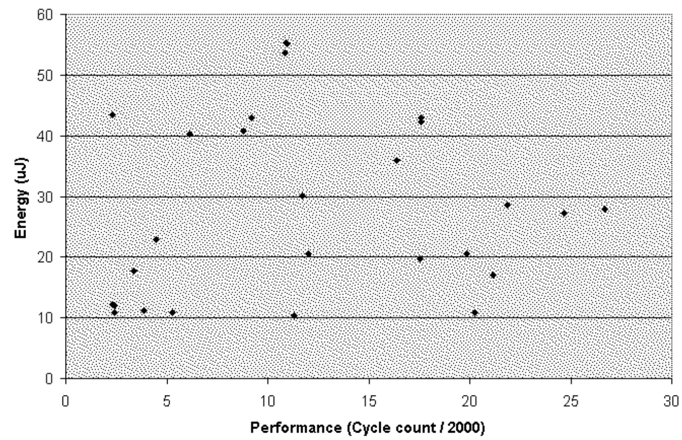

Figure 22. Energy Performance Tradeoff for integrate

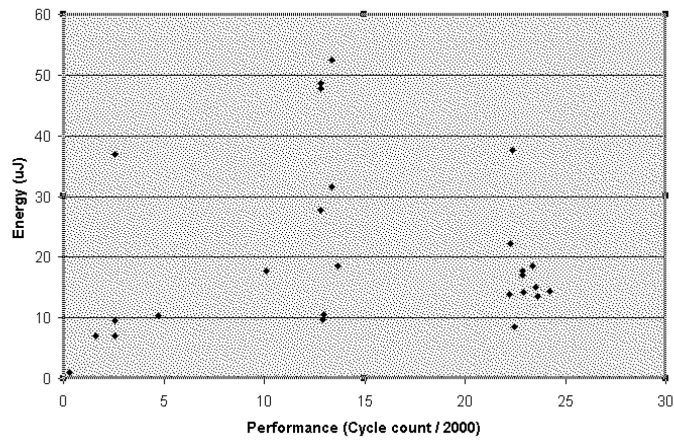

Figure 23. Energy Performance Tradeoff for Intpred

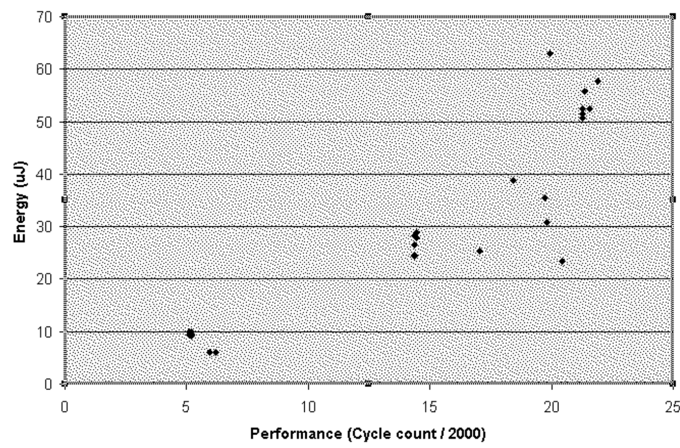

Figure 24. Energy Performance Tradeoff for Linear

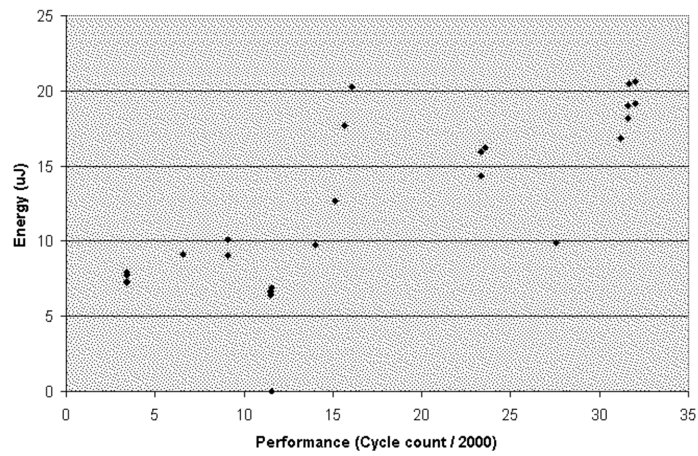

Figure 25. Energy Performance Tradeoff for Lineareqn

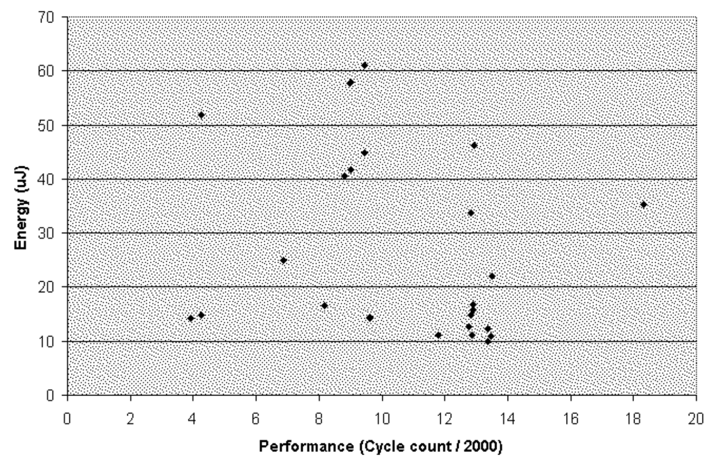

Figure 26. Energy Performance Tradeoff for Partpush

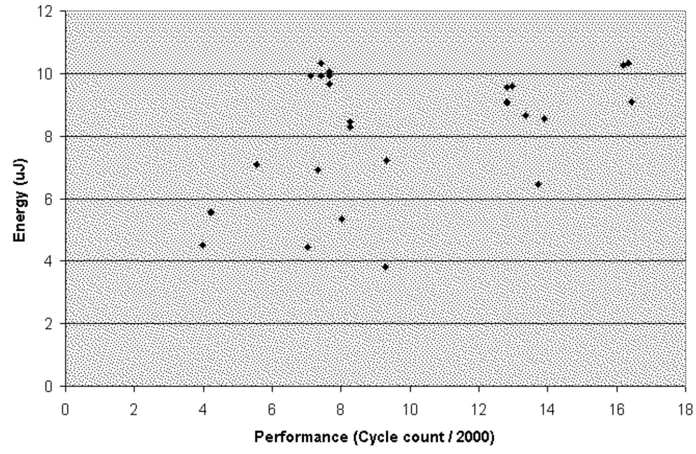

Figure 27. Energy Performance Tradeoff for Planc

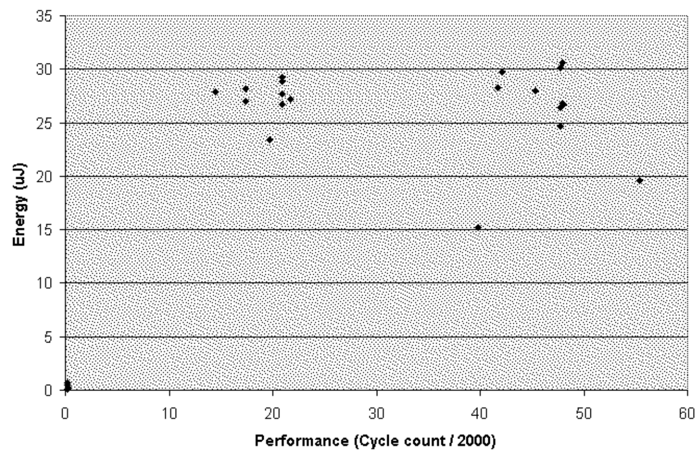

Figure 28. Energy Performance Tradeoff for Recurrence

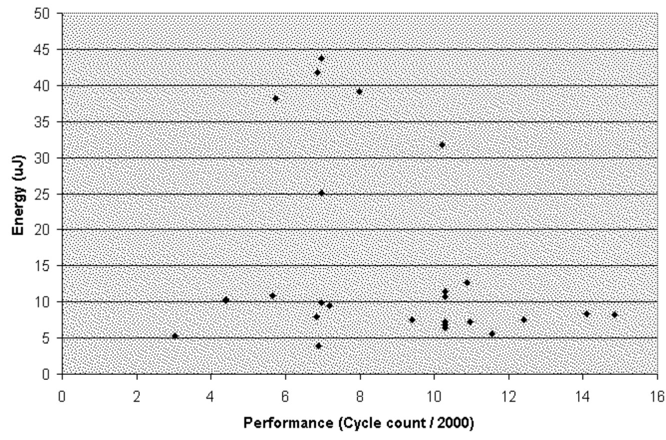

Figure 29. Energy Performance Tradeoff for Stateexcerpt

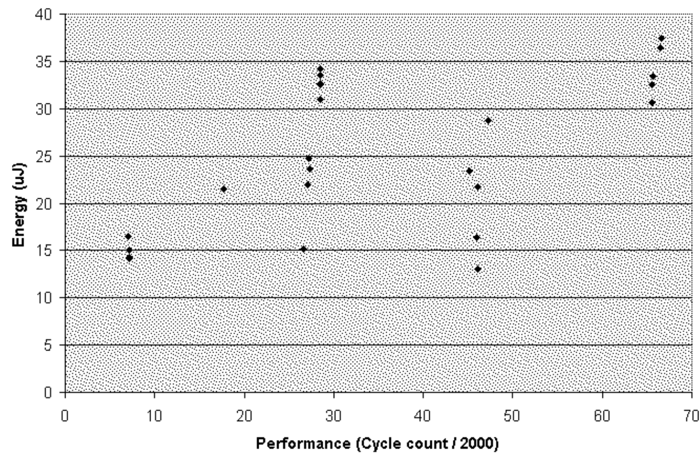

Figure 30. Energy Performance Tradeoff for Tridiag

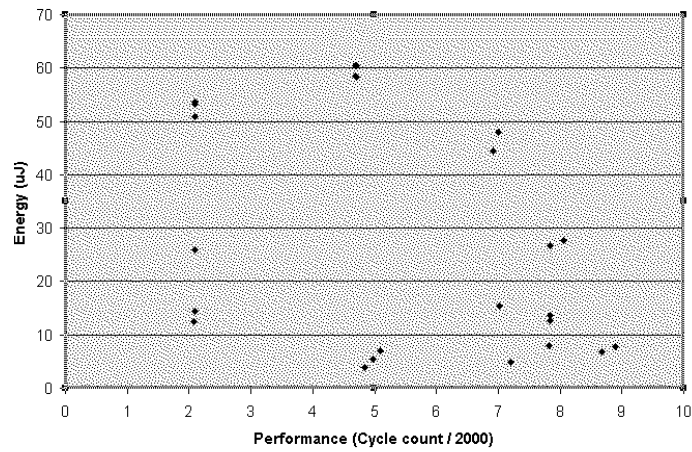

Figure 31. Energy Performance Tradeoff for Wavelet