Access Pattern-Based Memory and Connectivity Architecture Exploration

PETER GRUN, NIKIL DUTT, and ALEX NICOLAU
Center for Embedded Computer Systems, University of California

Memory accesses represent a major bottleneck in embedded systems power and performance. Traditionally, designers tried to alleviate this problem by relying on a simple cache hierarchy, or a limited use of special purpose memory modules such as stream buffers. Although real-life applications contain a large number of memory references to a diverse set of data structures, a significant percentage of all memory accesses in the application are generated from a few memory instructions that exhibit predictable, well-known access patterns; this creates an opportunity for memory customization, targeting the needs of these access patterns. We present APEX, an approach that extracts, analyzes and clusters the most active access patterns in the application, and aggressively customizes the memory architecture to match the needs of the application. Moreover, though the memory modules are important, the rate at which the memory system can produce the data for the CPU is significantly impacted by the connectivity architecture between the memory subsystem and the CPU. Thus, it is critical to consider the connectivity architecture early in the design flow, in conjunction with the memory architecture. We couple the exploration of memory modules together with their connectivity, to evaluate a wide range of cost, performance, and energy connectivity architectures. We use a heuristic to prune the design space, guiding the exploration towards the most promising designs. We present experiments on a set of large real-life benchmarks, showing significant performance improvements for varied cost and power characteristics, allowing the designer to evaluate customized memory and connectivity configurations for embedded systems.

Categories and Subject Descriptors: B.3.3 [**Hardware**]: Memory Structures—*Performance Analysis and Design Aids*

General Terms: Design, Experimentation, Performance

Additional Key Words and Phrases: Memory, architecture exploration, access patterns

1. MOTIVATION

In programmable embedded systems, memory represents a major performance and power bottleneck [Przybylski 1997]. Traditionally, designers have attempted to improve memory behavior by exploring different cache configurations, with limited use of more special purpose memory modules such as stream buffers [Jouppi 1990]. Real-life applications typically contain a large number of

This work was partially supported by grants from NSF (MIP-9708067), DARPA (F33615-00-C-1632) and a Motorola fellowship.
Authors' address: Center for Embedded Computer Systems, University of California, Irvine, CA, USA.

memory references to a diverse set of data structures; however, a significant percentage of all memory accesses in the application are generated from only a few instructions in the code. For instance, in vocoder (a GSM voice coding application with 15 KB lines of code) 62% of all memory accesses are generated by only 15 instructions. Furthermore, these instructions often exhibit well-known predictable access patterns. This presents a tremendous opportunity to customize the memory architecture to match the needs of the predominant access patterns in the application, and significantly improve the memory system behavior.

Moreover, the cost, bandwidth, and power footprint of the memory system is influenced not only by the memory modules employed, but also by the connectivity components (buses, interconnections) that transfer the data between the memory modules and the CPU. Indeed, though the memory configuration and characteristics are important, often the connectivity structure has a comparably large impact on the system performance, cost, and power, and considering it early in the design flow is crucial. In this article, we present a simulation/analysis approach that explores memory architectures (by extracting, analyzing, and clustering the most active memory access patterns in the application), and couples it with connectivity exploration (evaluating a wide range of connectivity configurations using components from a connectivity IP library, such as standard on-chip buses, MUX-based connections, and off-chip buses). This coupled approach improves the performance of the system, for varying cost, and power consumption, allowing the designer to evaluate and select custom memory configurations and architectures.

In our approach, we use the access patterns to customize the memory architecture, employing modules from a memory IP library, to explore a wide range of cost, performance, and power designs. We use a heuristic to prune the design space of such memory customizations, and guide the search towards the designs with best cost/gain ratios, exploring a space well beyond the one traditionally considered. We couple the memory exploration approach with the exploration of the connectivity architecture, to improve the behavior of the memory–connectivity system. There are two possible approaches to improve the memory system behavior: (a) A synthesis-oriented, optimization approach, where the result is a unique "best" solution, and (b) an exploration-based approach, where different memory system architectures are evaluated, and the most promising designs following a pareto-like shape are generated, allowing the designer to further refine the choice, according to the goals of the system. We follow the second approach: We guide the design space search towards the pareto points in different design spaces (such as the cost/performance, and performance/power spaces), pruning the noninteresting designs early in the exploration process, and avoiding full simulation of the design space.

In Section 2, we present related work in the area of connectivity and memory architecture exploration. In Section 3, we present the flow of our approach. In Section 4, we use an example application to illustrate our exploration strategy, and in Section 5 we show the details of our memory and connectivity exploration algorithm. We conclude with a set of experiments showing the cost, performance, and power trade-offs obtained by our coupled memory and connectivity exploration.

2. RELATED WORK

There has been related work in five main domains: (i) Disk file systems and databases, (ii) high-level synthesis, (iii) computer Architecture, (iv) programmable embedded systems, and (v) interface synthesis and layout/routing of the connectivity wires.

(i) In the domain of file systems and databases, there have been several approaches to use the file access patterns to improve the file system behavior. Parsons et al. [1997] present an approach allowing the application programmer to specify the file I/O parallel behavior using a set of templates which can be composed to form more complex access patterns. Patterson et al. [1995] advocate the use of hints describing the access pattern (currently supporting sequential accesses and an explicit list of accesses) to select particular prefetching and caching policies in the file system.

(ii) In the domain of high-level synthesis, custom synthesis of the memory architecture has been addressed for design of embedded ASICs. Catthoor et al. [1998] address memory allocation, packing the data structures according to their size and bitwidth into memory modules from a library, to minimize the memory cost, and optimize port sharing. Wuytack et al. [1996] present an approach to manage the memory bandwidth by increasing memory port utilization, through memory mapping and code reordering optimizations. Bakshi and Gajski [1995] present a memory exploration approach, combining memory modules using different connectivity and port configurations for pipelined DSP systems. We complement this work by extracting and analyzing the prevailing accesses in the application in terms of access patterns, their relationships, similarities and interferences, and customize the memory architecture using memory modules from a library to generate a wide range of cost/performance/power trade-offs in the context of programmable embedded systems. We use the connectivity and memory power/area estimation models from [Catthoor et al. 1998] to drive our exploration.

Narayan and Gajski [1994] synthesize the bus structure and communication protocols to implement a set of virtual communication channels, trading off the width of the bus and the performance of the processes communicating over it. Daveau et al. [1995] present a library-based exploration approach, where they use a library of connectivity components, with different costs and performance. We complement these approaches by exploring the connectivity design space in terms of all the three design goals: Cost, performance, and power simultaneously.

Givargis and Vahid [1998] present a connectivity exploration technique that employs different encoding techniques to improve the power behavior of the system. However, due to their platform-based approach, where they assume a predesigned architecture platform that they tune for power, they do not consider the cost of the architecture as a metric. Maguerdichian et al. [2001] present an on-chip bus network design methodology, optimizing the allocation of the cores to buses to reduce the latency of the transfers across the buses. Lahiri et al. [2000] present a methodology for the design

of custom system-on-chip communication architectures, which proposes the use of dynamic reconfiguration of the communication characteristics, taking into account the needs of the application.

(iii) In the domain of computer architecture, Jouppi [1990] and Palacharla and Kessler [1994] propose the use of hardware stream buffers to enhance the memory system performance. Reconfigurable cache architectures have been proposed recently [Veidenbaum et al. 1999] to improve the cache behavior for general purpose processors, targeting a large set of applications. However, the extra control needed for adaptability and dynamic prediction of the access patterns, although acceptable in general purpose computing where performance is the main target, may result in a power overhead, which is prohibitive in embedded systems that are typically power constrained. Instead of using such dynamic prediction mechanisms, we statically target the local memory architecture to the data access patterns.

On a related front, Hummel et al. [1994] address the problem of memory disambiguation in the presence of dynamic data structures to improve the parallelization opportunities. Instead of using this information for memory disambiguation, we use a similar type of closed form description generated by standard compiler analysis to represent the access patterns and guide the memory architecture customization.

(iv) In the domain of programmable embedded systems, Kulkarni [2001] and Panda et al. [1999] have addressed customization of the memory architecture targeting different cache configurations, or alternatively using on-chip scratch pad SRAMs to store data with poor cache behavior. Grun et al. [2001] present an approach that customizes the cache architecture to match the locality needs of the access patterns in the application. However, this work only targets the cache architecture, and does not attempt to use custom memory modules to target the different access patterns.

(v) Recent work on interface synthesis [Chou et al. 1995; Chung et al. 1996] presents techniques to formally derive node clusters from interface timing diagrams. These techniques can be used to provide an abstraction of the connectivity and memory module timings in the form of reservation tables [Hennessy and Patterson 1990]. Our algorithm uses the reservation tables [Grun et al. 1999] for performance estimation, taking into account the latency, pipelining, and resource conflicts in the connectivity and memory architecture.

At the physical level, a large body of work has addressed connectivity layout and wiring optimization and estimation. For instance, Chen et al. [1999] present a method to combine interconnect planning and floorplanning for deep submicron VLSI systems, where communication is increasingly important. Deng and Maly [2001] propose the use of a 2.5-D layout model, through a stack of single-layer monolithic ICs, to significantly reduce wire length. We use the area models presented in Chen et al. [2001] and Deng and Maly [2001] to drive our high-level connectivity exploration approach.

The work we present differs significantly from all the related work in that we aggressively analyze, cluster, and map memory access patterns to customized

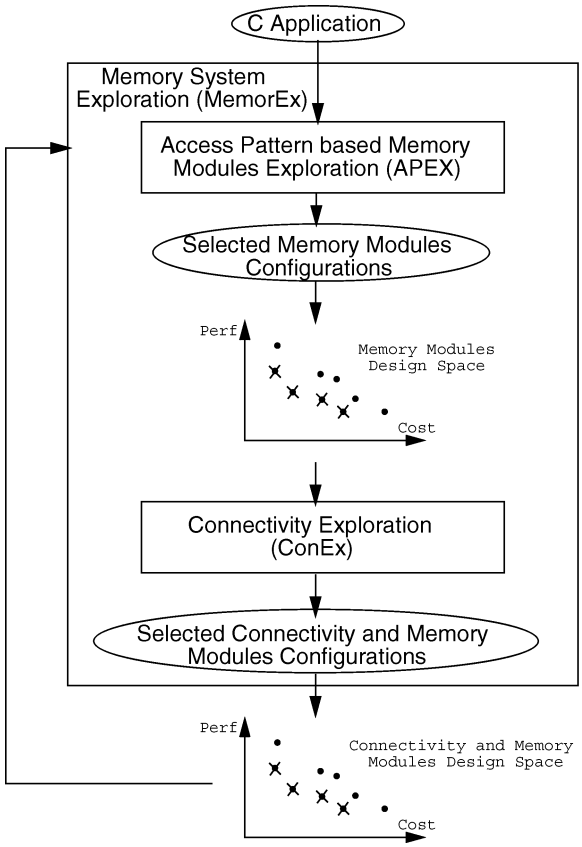

Fig. 1. The flow of our exploration approach.

memory architectures; this allows the designer to trade off performance and power against cost of the memory system.

Moreover, to our knowledge, none of the previous approaches has addressed connectivity exploration in conjunction with memory modules architecture, considering simultaneously the cost, performance, and power of the system, using a library of connectivity components including standard buses (such as AMBA, MUX-based connections, and off-chip buses). By pruning the noninteresting designs early in the design flow, and simulating only the most promising architectures, we allow the designer to explore the connectivity architectures space, to best trade off the different goals of the system.

3. OUR APPROACH

Figure 1 shows the flow of our approach. The connectivity exploration (ConEx) approach is part of the memorEx memory system exploration environment. Starting from the input application in C, our access pattern based memory exploration (APEX) algorithm first extracts the most active access patterns exhibited by the application data structures, and explores the memory module

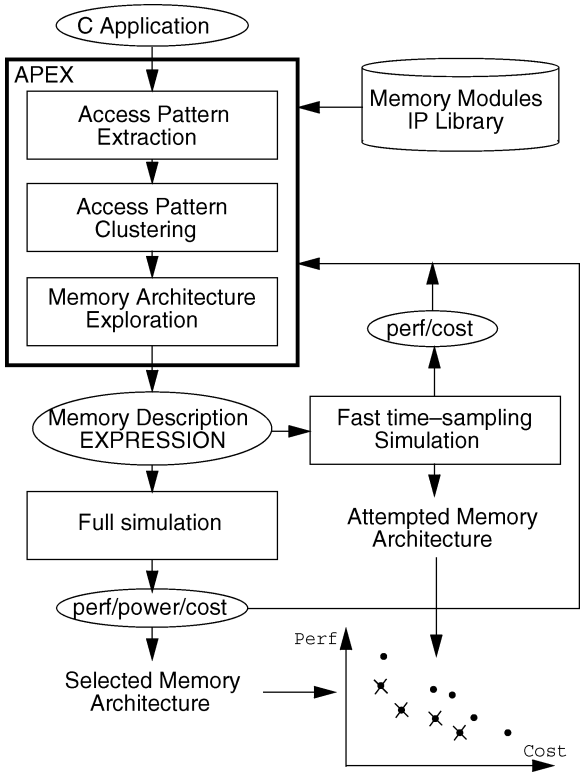

Fig. 2. The flow of our APEX approach.

configurations to match the needs of these access patterns; however, it assumes a simple connectivity model. Our ConEx connectivity exploration approach starts from this set of selected memory module configurations generated by APEX, and explores the most promising connectivity architectures that best match the performance, cost, and power goals of the system. Since the complete design space is very large, and evaluating all possible combinations in general is intractable, at each stage we prune out the noninteresting design configurations, and consider for further exploration only the points that follow a pareto-like curve shape in the design space.

Figure 2 presents the flow of our APEX approach. We start by extracting the most active access patterns from the input C application; we then analyze and cluster these access patterns according to similarities and interference, and customize the memory architecture by allocating a set of memory modules from a memory modules IP library. We explore the space of these memory customizations by using a heuristic to intelligently guide the search towards the most promising cost/performance memory architecture trade-offs. We prune the design space by using a fast time-sampling simulation to rule out the noninteresting parts of the design space, and then fully simulate and determine the power consumption only for the selected memory architectures. After narrowing down the search to the most promising cost/performance

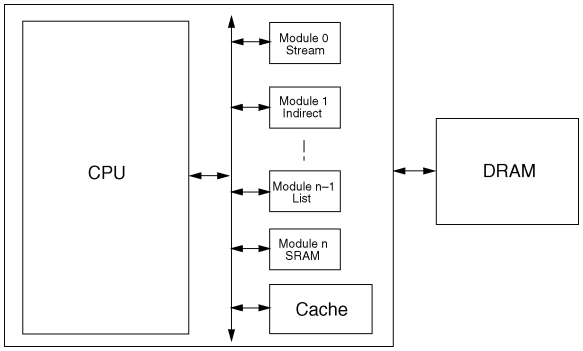

Fig. 3. Memory architecture template.

designs, we allow the designer to best match the power requirements of the system by providing full cost/performance/power characteristics for the selected designs.

The basic idea is to target specifically the needs of the most active memory access patterns in the application and customize a memory architecture, exploring a wide range of designs that exhibit varied cost, performance, and power characteristics.

Figure 3 presents the memory architecture template. The memory access requests from the processor are routed to one of the memory modules 0 through n or to the cache, based on the address. The custom memory modules can read the data directly from the DRAM, or alternatively can go through the cache which is already present in the architecture, allowing access patterns that exhibit locality to make use of the locality properties of the cache. The custom memory modules implement different types of access patterns, such as stream accesses, linked-list accesses, or a simple SRAM to store hard-to-predict or random accesses. We use custom memory modules to target the most active access patterns in the application, whereas the remaining, less frequent access patterns are serviced by the on-chip cache.

Starting from a memory architecture containing a set of memory modules, we map the communication channels between these modules, the off-chip memory, and the CPU to connectivity modules from a connectivity IP library. Figure 4(a) shows the connectivity architecture template for an example memory architecture, containing a cache, a stream buffer, an on-chip SRAM, and an off-chip DRAM. The communication channels between the on-chip memory modules, the off-chip memory modules, and the CPU can be implemented in many ways. One naive implementation is where each communication channel is mapped to one connectivity module from the library. However, though this solution may generate good performance, in general the cost is prohibitive. Instead, we cluster the communication channels into groups based on their bandwidth requirement, and map each such cluster to connectivity modules. Figure 4(b) shows an example connectivity architecture implementing the communication channels, containing two on-chip buses, a dedicated connection, and an off-chip bus.

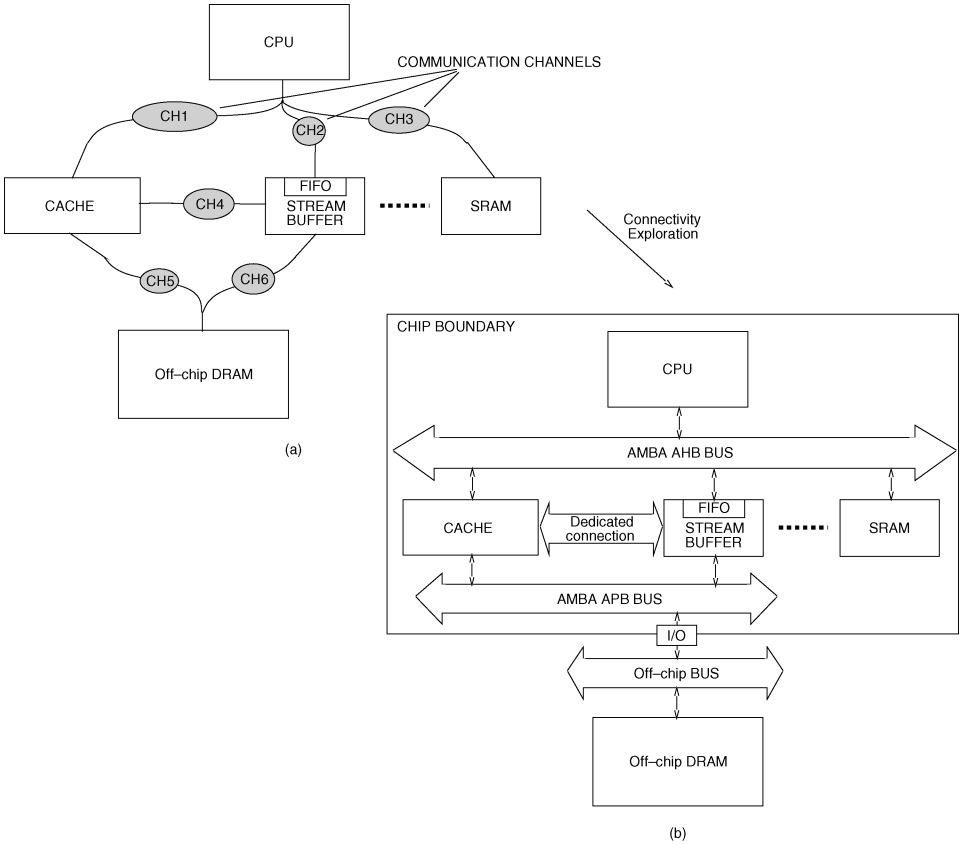

Fig. 4. (a) The connectivity architecture template and (b) an example connectivity architecture.

4. ILLUSTRATIVE EXAMPLE

We use the *compress* benchmark (from SPEC95) to illustrate the performance, power, and cost trade-offs generated by our approach. The benchmark contains a varied set of access patterns, presenting interesting opportunities for customizing the memory architecture.

4.1 Memory Exploration

We start by profiling the application, to determine the most active basic blocks and memory references. In the compress benchmark, 40% of all memory accesses are generated by only 19 instructions. Indeed, this is a typical situation: In many large real-life applications, a significant percentage of the memory accesses are generated from a few instructions in the code.

By traversing the most active basic blocks, we extract the most active access patterns from the application. Figure 5 shows an excerpt of code from compress, containing references to three arrays: htab, codetab, and rmask. htab is a hashing table represented as an array of 69,001 unsigned longs (we assume that both longs and ints are stored on 32 bits), codetab is an array of 69,001

```
while ( ... )
  ...
  ... = htab[code];
  code = codetab[code];
  ...
while ( ... )
  ...
  ... = rmask[r_off]
  ...

Access patterns:
    ap1 = htab[ap2]
    ap2 = codetabl[ap2]
    ap3 = rmask[unknown]
```

Fig. 5. Example access patterns.

shorts, and rmask is an array of nine characters. The sequence of accesses to htab, codetab, and rmask represents access patterns ap1, ap2, and ap3, respectively. The hashing table htab is traversed using the array codetab as an indirect index, and the sequence of accesses to the array codetab is generated by a self-indirection, by using the values read from the array itself as the next index. The sequence in which the array rmask is traversed is difficult to predict, due to a complex index expression computed across multiple functions. Therefore we consider the order of accesses as unknown. However, rmask represents a small table of coefficients, accessed very often.

Compress contains many other memory references exhibiting different types of access patterns such as streams with positive or negative stride. We extract the most active access patterns in the application and cluster them according to similarity and interference. Since all the access patterns in a cluster will be treated together, we group together the access patterns that are compatible (for instance access patterns that are similar and do not interfere) in the hope that all the access patterns in a cluster can be mapped to one custom memory module.

Next, for each such access pattern cluster, we allocate a custom memory module from the memory modules library. We use a library of parameterizable memory modules containing both generic structures such as caches and on-chip SRAMs, as well as a set of parameterizable custom memory modules developed for specific types of access patterns such as streams with positive, negative, or nonunit strides, indirect accesses, self-indirect accesses, and linked-list accesses. Although these custom memory modules themselves are not the contribution of this article (we simply use them as input to our memory architecture exploration algorithm), we briefly describe one such module for illustration. The custom memory modules are based on approaches proposed in the general purpose computing domain [Chiueh 1994], with the modification that the dynamic prediction mechanisms are replaced with the static compile-time analysis of the access patterns, and the prefetched data is stored in special purpose FIFOs.

For instance, for the example access pattern ap2 from compress, we use a custom memory module implementing a self-indirect access pattern, whereas

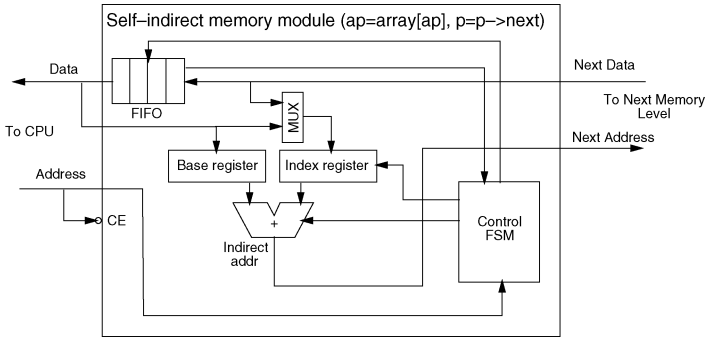

Fig. 6. Self-indirect custom memory module.

for the access pattern ap3, due to the small size of the array rmask, we use a small on-chip SRAM [Panda et al. 1999]. Figure 6 presents an outline of the self-indirect custom memory module architecture used for the access pattern ap2, where the value read from the array is used as the index for the next access to the array. The base register stores the base address of the array, the index register stores the previous value that will be used as an index in the next access, and the small FIFO stores the stream of values read from the next memory level, along with the address tag used for write coherency. When the CPU sends a read request, the data is provided from the FIFO. The empty spot in the FIFO initiates a fetch from the next level memory to bring in the next data element. The adder computes the address for the next data element based on the base address and the previous data value. We assume that the base register is initialized to the base of the codetab array and the index register to the initial index through a memory mapped control register model (a store to the address corresponding to the base register writes the base address value into the register).

The custom memory modules from the library can be combined together, based on the relationships between the access patterns. For instance, the access pattern ap1 uses the access pattern ap2 as an index for the references. In such a case, we use the self-indirection memory module implementing ap2 in conjunction with a simple indirection memory module, which computes the sequence of addresses by adding the base address of the array htab with the values produced by ap2, and generate ap1 = htab[ap2].

After selecting a set of custom memory modules from the library, we map the access pattern clusters to memory modules. Starting from the traditional memory architecture, containing a small cache, we incrementally customize access pattern clusters, to significantly improve the memory behavior. Many such memory module allocations and mappings are possible. Exploring the full space of such designs would be prohibitively expensive. To provide the designer with a spectrum of such design points without the time penalty of investigating the full space, we use a heuristic to select the most promising memory architectures, providing the best cost/performance/power trade-offs.

For the compress benchmark, we explore the design space choosing a set of five memory architectures that provide advantageous cost/performance

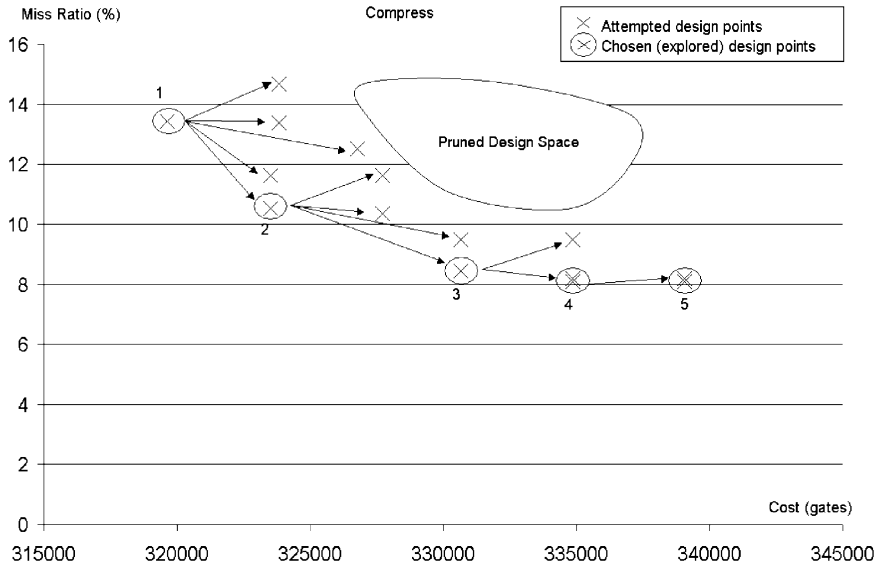

Fig. 7. The most promising memory modules architectures for the compress benchmark.

trade-offs. Figure 7 shows the memory modules architectures explored by APEX for the compress example. The x-axis represents the cost of the memory modules in basic gates, and the y-axis represents the overall miss ratio.[1] The overall miss rate of the memory system is reduced by 39%, generating a significant performance improvement for varied cost and power characteristics. APEX prunes the noninteresting designs, on the inside of the pareto curve, choosing only the most promising cost/performance architectures for further exploration. The points labeled 1 through 5 represent the selected memory modules designs that will be used as the starting point for the connectivity exploration.

In this manner, we can customize the memory architecture by extracting and analyzing the access patterns in the application, thus substantially improving the memory system behavior, and allowing the designer to trade off the different goals of the system.

4.2 Connectivity Exploration

In the previous section, we illustrated our APEX, generating a set of promising memory modules architectures. For each such memory modules architecture, a set of different connectivity architectures are possible, each resulting in different cost, performance, and power characteristics. Our ConEx approach starts from the memory modules architectures generated by APEX, and explores the connectivity configurations using components from a connectivity library (such as the AMBA buses, MUX-based connections, etc.), trading off the

[1]We define hits to be accesses to on-chip memories (such as caches or SRAMs) and misses as accesses to off-chip memories.

cost, performance, and power for the full memory system, taking into account both the memory modules and the connectivity structure.

For our compress illustrative example benchmark, APEX selects the most promising memory modules configurations. The resulting memory architectures employ different combinations of modules such as caches, SRAMs, and DMA-like custom memory modules storing well-behaved data such as linked lists, arrays of pointers, streams, etc. [Grun et al. 2001]. Figure 7 shows the memory modules architectures explored by APEX for the compress example. The points labeled 1 through 5 represent the selected memory modules designs that will be used as the starting point for the connectivity exploration.

Each such selected memory architecture may contain multiple memory modules with different characteristics and communication requirements. For each such architecture, different connectivity structures with varied combinations of connectivity modules from the library may be used. For instance, the memory modules architecture labeled 3 in Figure 7 contains a cache, a memory module for stream accesses, a memory module for self-indirect[2] array references, and an off-chip DRAM. When using dedicated or MUX-based connections from the CPU to the memory modules, the latency of the accesses is small, at the expense of longer connection wires. Alternatively, when using a bus-based connection, such as the AMBA system bus (ASB), the wire length decreases, at the expense of increased latency due to the need for more complex arbitration. Similarly, when using wider buses, with pipelined or split transaction accesses, such as the AMBA high-performance bus (AHB), the wiring and bus controller area increases further. Moreover, all these considerations impact the energy footprint of the system. For instance, longer connection wires generate larger capacitances, which may lead to increased power consumption.

Figure 8 shows the ConEx connectivity exploration for the compress benchmark. The x-axis represents the cost of the memory and connectivity system. The y-axis represents the average memory latency, including the latency due to the memory modules, as well as the latency due to the connectivity. The average memory latency is reduced from 10.6 cycles to 6.7 cycles, representing a 36% improvement,[3] while trading off the cost of the connectivity and memory modules.

Alternatively, for energy-aware designs, similar trade-offs are obtained in the cost/power or the performance/power design spaces (the energy consumption trade-offs are presented in Section 6). In this manner, we can customize the connectivity architecture, thus substantially improving the memory and connectivity system behavior, and allowing the designer to trade off the different goals of the system.

[2]We call *self-indirect* the array references that use the current array element value to compute the index for the next array element access.

[3]Please note that to keep the figures clear, we did not include the uninteresting designs exhibiting very bad performance (many times worse than the best designs). Although those designs would increase the performance variation even further, in general they are not useful.

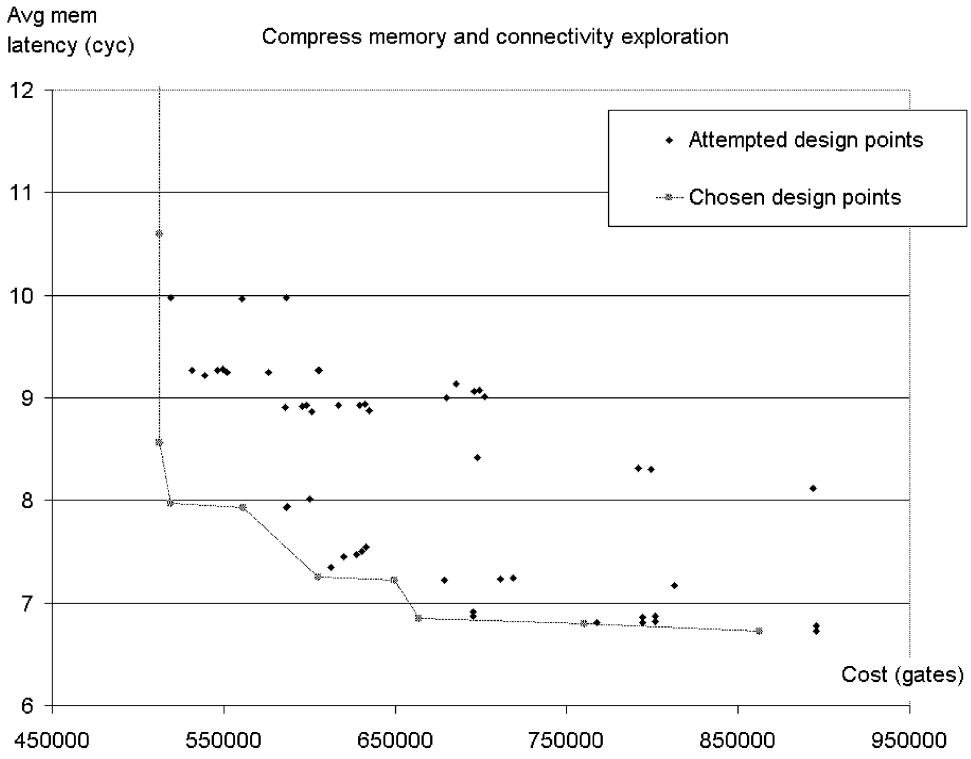

Fig. 8. The connectivity architecture exploration for the compress benchmark.

5. THE MEMORY AND CONNECTIVITY EXPLORATION APPROACH

5.1 The Access Pattern-Based Memory Exploration (APEX) Approach

Our access pattern-based memory exploration (APEX) approach is a heuristic method to extract, analyze, and cluster the most active access patterns in the application, and customize the memory architecture, explore the design space to trade off the different goals of the system. It contains two phases: (I) access pattern clustering; and (II) exploration of custom memory configurations.

5.1.1 *Access Pattern Clustering.* In the first phase of our approach, we extract the access patterns from the application, analyze and group them into access pattern clusters, according to their relationships, similarities, and interferences. Figure 9 presents an outline of the access pattern extraction and clustering algorithm. The access pattern clustering algorithm contains four steps.

(1) We extract the most active access patterns from the input application. We consider three types of access patterns: (a) access patterns that can be determined automatically by analyzing the application code; (b) access patterns about which the user has prior knowledge; and (c) access patterns that are difficult to determine, or are input-dependent.

```
Procedure GenerateAccessPatternClusters
Input: Application in C and Access Pattern Assertions
Output: Access Pattern Clusters
begin
  1. Extract Access Patterns from application
  2. Build Access Pattern Graph APG(AP,Arcs)
  3. Build Access Pattern Compatibility Graph
       APCG(AP,CompatibilityArcs)
  4. Choose Cliques Of Compatibility Arcs to form
       Access Pattern Clusters
end
```

Fig. 9. Access Pattern Clustering algorithm.

(a) Often access patterns can be determined at compile time, using traditional compiler analysis. Especially in DSP and embedded systems, the access patterns tend to be more regular and predictable at compile time (e.g., in video, image, and voice compression).

First, we use profiling to determine the most active basic blocks in the application. For each memory reference in these basic blocks, we traverse the use-def chains to construct the address expression until we reach statically known variables, constants, loop indexes, or other access patterns. This closed form formula represents the access pattern of the memory reference. If all the elements in this expression are statically predictable, and the loop indexes have known bounds, the access pattern represented by this formula is predictable.

(b) In the case of well-known data structures (e.g., hashing tables, linked lists, etc.), or well-understood high-level concepts (such as the traversal algorithms in well-known DSP functions), the programmer has prior knowledge on the data structures and the access patterns. By providing this information in the form of assertions, he can give hints on the predominant accesses in the application. Especially when the memory references depend on variables that traverse multiple functions, indirections, and aliasing, and determining the access pattern automatically is difficult, allowing the user to input such readily available information, significantly improves the memory architecture customization opportunities.

(c) In the case of memory references that are complex and difficult to predict, or depend on input data, we treat them as random access patterns. Whereas for such references it is often impossible to fully understand the access pattern, it may be useful to use generic memory modules such as caches or on-chip scratch pad memories to exploit the locality trends exhibited. A detailed description of the access pattern clustering algorithm is presented in Grun et al. [2001].

(2) In the second step of the access pattern clustering algorithm we build the access pattern graph (APG), containing as vertices the most active access patterns from the application. The arcs in the APG represent properties such as similarity, interference, whether two access patterns refer to the same data structure, or whether an access pattern uses another access pattern as an index for indirect addressing, or pointer computation.

```
Procedure Exploration
Input: Access Pattern Clusters, and the Memory Modules Library
Output: The Memory Architecture design points w/ best cost/perf ratios
begin
  Initialize the memory architecture to contain the initial_cache
  While cost of memory architecture < cost_constraint do
    While cost of memory architecture < cost_constraint and
      more allocations and mappings possible do
      For all access pattern clusters sharing a memory module
        Allocate a memory module and map the cluster to it
        Estimate cost of new memory architecture
        If (cost of new memory architecture > cost_constraint) continue
        Estimate performance of new memory architecture (time-sampling)
        Save current memory architecture alternative
        Undo memory module allocation and mapping
      end
      Choose the memory architecture with best cost/performance
      Perform full simulation of new design point
    end
    Double the cache size
  end
end
```

Fig. 10. Exploration algorithm.

(3) Based on the APG, we build the access pattern compatibility graph (APCG), which has the same vertices as the APG (the access patterns), but the arcs represent compatibility between access patterns. We say two access patterns are *compatible* if they can belong to the same access pattern cluster. For instance, access patterns that are similar (e.g., both have stream-like behavior), but which have little interference (are accessed in different loops) may share the same custom memory module, and it makes sense to place them in the same cluster. The meaning of the access pattern clusters is that all the access patterns in a cluster will be allocated to one memory module.
(4) In the last step of the access pattern clustering algorithm, we find the cliques of fully connected subgraphs in the APCG compatibility graph. Each such clique represents an access pattern cluster, where all the access patterns are compatible, according to the compatibility criteria determined from the previous step (for a complete description of the compatibility criteria, please refer to Grun et al. [2001]). Each such access pattern cluster will be mapped in the following phase to a memory module from the library.

5.1.2 *Exploring Custom Memory Configurations.* In the second phase of the APEX approach, we explore the custom memory module implementations and access pattern cluster mappings, using a heuristic to find the most promising design points.

Figure 10 presents an outline of our exploration heuristic. We first initialize the memory architecture to contain a small traditional cache, representing the starting point of our exploration.

For each design point, the number of alternative customizations available is large, and fully exploring them is prohibitively expensive. For instance, each access pattern cluster can be mapped to custom memory modules from the library,

or to the traditional cache, each such configuration generating a different cost/performance/power trade-off. To prune the design space, at each exploration step we first estimate the incremental cost and gain obtained by the further possible customization alternatives, then choose the alternative leading to the best cost/gain ratio for further exploration. Once a customization alternative has been chosen, we consider it the current architecture, and perform full simulation for the new design point. We then continue the exploration, by evaluating the further possible customization opportunities, starting from this new design point.

We tuned our exploration heuristic to prune out the design points with poor cost/ performance characteristics, guiding the search towards points on the lower bound of the cost/performance design space.

For performance estimation purposes, we use a time-sampling technique, which significantly speeds the simulation process. Although this may not be highly accurate compared to full simulation, the fidelity is sufficient to make good incremental decisions guiding the search through the design space. To verify that our heuristic guides the search towards the pareto curve of the design space, we compare the exploration results with a full simulation of all the allocation and access pattern mapping alternatives for a large example. Indeed, as shown in Section 6, our algorithm finds the best cost/performance points in the design space, without requiring full simulation of the design space. For more details on our APEX algorithm, please refer to Grun et al. [2001].

5.2 Connectivity Exploration Algorithm

Our connectivity exploration (ConEx) algorithm is a heuristic method to evaluate a wide range of connectivity architectures, using components from a connectivity IP library, and selecting the most promising architectures, which best trade-off the connectivity cost, performance, and power.

Figure 11 shows our ConEx algorithm. The input to our ConEx algorithm is the application in C, a set of selected memory modules architectures (generated by the APEX exploration step [Grun et al. 2001]), and the connectivity library. Our algorithm generates as output the set of most promising connectivity/memory modules architectures, in terms of cost, performance, and power.

For each memory modules architecture selected in the APEX memory modules exploration stage [Grun et al. 2001], multiple connectivity implementations are possible. Starting from these memory modules architectures, we explore the connectivity configurations by taking into account the behavior of the complete memory and connectivity system, allowing the designer to trade off the cost, performance, and power of the design. The ConEx algorithm proceeds in two phases: (I) evaluate connectivity configurations and (II) select most promising designs.

(I) *Evaluate connectivity configurations.* For each memory architecture selected from the previous APEX memory modules exploration phase [Grun et al. 2001], we evaluate different connectivity architecture templates and connectivity allocations using components from the connectivity IP library. We estimate the cost, performance, and power of each such connectivity architecture, and

```
Algorithm ConEx
Input: C Application, Selected Memory Modules Architectures
Output: The Combined Memory Modules and Connectivity Design Points
  w/ best cost/performance/power trade-offs
begin
  Phase I:
    combined_design_points = φ
    For each selected memory module architecture mem_arch
      connect_design_points = ConnectivityExploration(mem_arch)
      Select the local most promising connectivity desing points from
        connect_design_points
      Add selected design points to combined_design_points
  Phase II:
    Simulate the design points from combined_design_points
    Select the global most promising combined memory modules and connectivity
      design points from combined_design_points
end

Procedure ConnectivityExploration(Memory Modules Architecture mem_arch)
Input: The C Application and the Memory Modules Architecture mem_arch,
  the Connectivity Library
Output: The most promising Connectivity Design Points
begin
  Profile the Memory Modules Architecture mem_arch
  Construct the Bandwidth Requirement Graph (BRG)
  Allocate each arc in the BRG to a logical connection cluster
  connect_design_points = φ
  do{
    if number_of_logical_connections ≤ max_cost_constraint
      Allocate the logical connections to physical connections
        from the Connectivity Library
      Estimate the Cost, Performance and Power of connecitivity architecture
      Add this connectivity architecture to connect_design_points
    Merge the two logical connection clusters with lowest bandwidth requirement
      hierarchycally into a larger cluster
  }while(more clusters can be merged)
  return connect_design_points;
end
```

Fig. 11. Connectivity exploration algorithm.

perform an initial selection of the most promising design points for further evaluation.

We start by profiling the bandwidth requirement between the memory modules and CPU for each memory modules architecture selected from APEX, and constructing a bandwidth requirement graph (BRG). The bandwidth requirement graph represents the bandwidth requirements of the application for the given memory modules architecture. The nodes in the BRG represent the memory and processing cores in the system (such as the caches, on-chip SRAMs, DMAs, off-chip DRAMs, the CPU, etc.), and the arcs represent the channels of communication between these modules. The BRG arcs are labeled with the average bandwidth requirement between the two modules.

Each arc in the BRG has to be implemented by a connectivity component from the connectivity library. One possible connectivity architecture is where

each arc in the BRG is assigned to a different component from the connectivity library. However, this naive implementation may result in excessively high cost, because it does not try to share the connectivity components. To allow different communication channels to share the same connectivity module, we hierarchically cluster the BRG arcs into logical connections, based on the bandwidth requirement of each channel. We first group the channels with the lowest bandwidth requirements into logical connections. We label each such cluster with the cumulative bandwidth of the individual channels, and continue the hierarchical clustering. For each such clustering level, we then explore all feasible assignments of the clusters to connectivity components from the library, and estimate the cost, performance, and power of the memory and connectivity system.

(II) *Select most promising designs.* In the second phase of our algorithm, for each memory and connectivity architecture selected from phase I, we perform full simulation to determine accurate performance and power metrics. We then select the best combined memory and connectivity candidates from the simulated architectures.

Whereas in phase I, we selected separately for each memory module architecture the best connectivity configurations, in phase II we combine the selected designs and choose the best overall architectures, in terms of both the memory module and connectivity configuration.

The different design points present different cost, performance, and power characteristics. In general, these three optimization goals are incompatible. For instance, when optimizing for performance, the designer has to give up either cost or power. Typically, the pareto points in the cost/performance space have a poor power behavior, whereas the pareto points in the performance/power space will incur a large cost. We select the most promising architectures using three scenarios: (a) in a power-constrained scenario, where the energy consumption has to be less then a threshold value, we determine the cost/performance pareto points, to optimize for cost and performance, while keeping the power less then the constraint; (b) in a cost-constrained scenario, we compute the performance/power pareto points; and (c) in a performance-constrained scenario, we compute the pareto points in the cost–power space, optimizing for cost and power, while keeping the performance within the requirements.

(a) In the power-constrained scenario, we first determine the pareto points in the cost–performance space. A design is on the pareto curve if there is no other design which is better in both cost and performance. We then collect the energy consumption information for the selected designs. The points on the cost–performance pareto curve may not be optimal from the the energy consumption perspective. From the selected cost–performance pareto points we choose only the ones that satisfy the energy consumption constraint. The designer can then trade off the cost and performance of the system to best match the design goals.
(b) In the cost-constrained scenario, we start by determining the pareto points in the performance–power space, and use the system cost as a constraint. Conversely, the pareto points in the performance–power space are in general not optimal from the cost perspective.

(c) When using the performance as a constraint, we determine the cost–power pareto points.

For performance and power estimation purposes, we use a time-sampling technique [Kessler et al. 1991], which significantly speeds the simulation process. Although this may not be highly accurate compared to full simulation, the fidelity is sufficient to make good incremental decisions guiding the search through the design space. To verify that our heuristic guides the search towards the pareto curve of the design space, we compare the exploration results with a full simulation of all the memory and connectivity mapping alternatives for two large examples. Indeed, as shown in Section 6, our algorithm successfully finds the best points in the design space, without requiring full simulation of the design space.

5.2.1 *Cost, Performance, and Power Models.* We present in the following the cost, performance and power models used during our memory modules and connectivity exploration approach.

The cost of the chip is composed of two parts: the cost of the cores (such as CPU cores, memories, and controllers), and the cost of the connectivity wiring. We use the method employed in Chen et al. [1999] to compute the total chip area: Since the wiring area and the cores area can be proportionally important, we use two factors α and β tuned so that the overall wire and core areas are balanced [Chen et al. 1999]

$$Chip\_area = \alpha * Wire\_area + \beta * Cores\_area$$

where the *Wire_area* is the area used by the connectivity wires, and the *Cores_area* is the area of the memory modules, memory controllers, CPU cores, and bus controllers.

For the connectivity cost we consider the wire length and bitwidth of the buses, and the complexity of the bus controller. We estimate the wire length to half the perimeter of the modules connected by that wire [Catthoor et al. 1998]:

$$Conn\_length = \Sigma(2 * \sqrt{module\_area})$$

where the sum is over all the modules connected by that connectivity component, and the *module_area* is the area of each such module. The area of this connectivity is then

$$Conn\_area = \alpha * Conn\_length * Conn\_bitwidth + \beta * Controller\_area$$

where *Conn_bitwidth* is the connectivity bitwidth, *Controller_area* is the area of the connectivity controller, and α and β are the two scaling factors determined as mentioned above.

We determine the cost of the on-chip cache using the cost estimation techniques presented in Catthoor et al. [1998]. For the cost of the custom memory modules explored in the previous stage of our DSE approach [Grun et al. 2001] we use figures from the Synopsys Design Compiler. For the CPU area (used to compute the wire lengths for the wires connecting the memory modules to the CPU), we use figures reported for the LEON SPARC gate count in 0.25 um [Givargis and Vahid 1998].

Since the area of the off-chip memory is not as important as for the on-chip components, we do not consider the off-chip memory area into our cost function.

We compute the performance of the memory system by generating a memory simulator for the specific memory modules and connectivity architectures [Mishra et al. 2001]. We describe the timings and pipelining of the memory and connectivity components using reservation tables, as presented in [Grun et al. 1999]. Buses may support continuous or split transactions, with different levels of pipelining. These features are also captured using the reservation tables model, augmented with explicit information on the presence of split or continuous transactions.

The memory system energy consumption comprises two parts: the connectivity energy consumption, and the memory modules energy consumption.

We estimate the energy consumption of the connectivity components based on the power estimation technique presented in Catthoor et al. [1998]:

$$Econn/access = 1/2 * Bus\_bitwidth * Atoggle * Fclock * (Cdriver + Cload) * Vdd^2 * access\_latency$$

where $Econn/access$ is the energy per access consumed by the connectivity module, $Bus\_bitwidth$ is the bitwidth of the bus, $Atoggle$ is the probability that a bit line toggles between two consecutive transfers, $Fclock$ is the clock frequency, and $Cdriver$ and $Cload$ are the capacitance of the buffer that drives the connectivity, and the total load capacitance of the wires, respectively.

We compute the load capacitance of the on-chip interconnect as

$$Cload = Lconn * Cmm,$$

where $Lconn$ is the length of the connectivity (computed as described above), and Cmm is the capacitance per mm of the wires. We assume a capacitance of 0.19 pF/mm for 0.25 um technology [Catthoor et al. 1998]. Although this figure is for the first metal layer, the capacitance values for the different layers are not dramatically different [Catthoor et al. 1998].

For the driver capacitance we use the approach presented in Liu and Svenson [1994]. Assuming that the size ratio in the inverter chain is 4, and the inverter that drives the load has a capacitance of 1/4 of its load, the total capacitance of the buffer is about 30% of the total load. The total capacitance ratio of the inverter chain is

$$\frac{1}{4} + \frac{1}{16} + \frac{1}{64} + \cdots = 0.3 \quad Cdriver = Cload * 0.3.$$

We compute the cache energy consumption per access using CACTI [Reinman and Jouppi 1999]. We determine the energy consumed by off-chip accesses, including the off-chip DRAM power, I/O pins, and assuming 30 mm off-chip bus length [Catthoor et al. 1998]. For the off-chip connectivity, we use the capacitance figures presented in Catthoor et al. [1998]: A typical off-chip bus capacitance is 0.1 pF/mm (we assume a 30 mm off-chip bus), and the bus driver capacitance is 5 pF. The chip I/O pins capacitance varies between 1 pF and 10 pF depending on the packaging type (we assume a capacitance of 5 pF for the I/O pins).

For the off-chip DRAM energy consumption there is a lot of variation among the figures considered by different researchers [Catthoor et al. 1998; Hicks et al. 1997; Vijaykrishnan et al. 2000], depending on the main memory type technology. The ratio between the energy consumed by on-chip cache accesses and off-chip DRAM accesses varies significantly (Catthoor et al. [1998] reports a ratio of 3 to 5 for accesses of same size, and Hicks et al. [1997] reports a ratio between one and two orders of magnitude; however, it is not clear if this includes the connectivity energy). To keep our technique independent of such technology figures, and allow the designer to determine the relative importance of these factors, we define a ratio R:

$$R = Emain\_memory\_access/Ecache\_access$$

where $E\_main\_memory\_access$ and $Ecache\_access$ are the energy consumed per access by the main memory and the cache, respectively, for accesses of the same size. We assume a ratio of 5, compared to an 8k two-way set associative cache.

We assume that the energy consumed by the custom memory controllers presented in Grun et al. [2001] to be similar to the energy consumed by the cache controller.

5.2.2 *Coupled Memory/Connectivity Exploration Strategy.* The quality of the final selected memory–connectivity architectures in different spaces, such as cost/performance, or cost/power spaces, depends on the quality of the initial memory modules architectures selected as starting points for the connectivity exploration. The memory modules architecture selection has to be driven by the same metric as the connectivity architecture selection. For instance, when cost and performance are important, we guide the search towards the cost/performance pareto points both in the early APEX memory modules exploration, as well as in the ConEx connectivity exploration, and use power as a constraint. Alternatively, when cost and power are important, we use cost/power as the metric to guide both the APEX and the ConEx explorations. For this, we modified the APEX [Grun et al. 2001] algorithm to use cost/power as the exploration driver, to determine the cost/power pareto points for the memory modules architectures. We then use these architectures as the starting point for the connectivity exploration.

To verify the sensitivity of the exploration on the memory modules architectures used as the starting point, we compare three exploration strategies, using different sets of starting memory modules architectures: (i) pruned exploration, where we select the most promising memory modules and connectivity architectures, and perform full simulation to determine the pareto curve without fully exploring the design space; (ii) neighborhood exploration, where we expand the design space by including also the points in the neighborhood of the architectures selected in the pruned approach; and (iii) full space exploration, the naive approach where we fully simulate the design space, and compute the pareto curve.

(i) In the pruned exploration approach, we start by selecting the most promising memory modules configurations, and use them as input for the connectivity exploration phase. For each such memory module architecture, we

then explore different connectivity designs, estimating the cost, performance, and energy consumption, and selecting at each step the best cost, performance, and power trade-offs. We then simulate only the selected designs and determine the pareto points from this reduced set of alternatives, in the hope that we find the overall pareto architectures, without fully simulating the design space.

(ii) To increase the chances of finding the designs on the actual pareto curve, we expand the explored design space by including the memory modules architectures in the neighborhood of the selected designs. In general, this leads to more points in the neighborhood of the pareto curve being evaluated, and possibly selected.

(iii) We compare our pruned and neighborhood exploration approaches to the brute force approach, where we fully simulate the design space and fully determine the pareto curve. Clearly, performing full simulation of the design space is very time consuming and often intractable. We use the naive full space exploration approach only to verify that our pruned and neighborhood exploration strategies successfully find the pareto curve designs points, while significantly reducing the computation time.

Clearly, by intelligently exploring the memory modules and connectivity architectures using components from a library, it is possible to explore a wide range of memory system architectures, with varied cost, performance, and power characteristics, allowing the designer to best trade off the different goals of the system. We successfully find the most promising designs following the pareto-like curve, without fully simulating the design space.

6. EXPERIMENTS

We performed a set of experiments on a number of large multimedia and scientific applications to show the performance, cost, and power trade-offs generated by our approach. Our exploration algorithm guides the search towards the points on the pareto curve[4] of the design space, pruning out the noninteresting designs. To verify that our design space exploration (DSE) approach successfully finds the points on the pareto curve, we compare the exploration algorithm results with the actual pareto curve obtained by fully simulating the design space.

6.1 Memory Exploration

6.1.1 *Experimental Setup.* We simulated the design alternatives using our simulator based on the SIMPRESS [Mishra et al. 2001] memory model, and SHADE [Cmelik and Keppel 1993]. We assumed a processor based on the SUN SPARC,[5] and we compiled the applications using gcc. We estimated the cost of the memory architectures (we assume the cost in equivalent basic gates)

[4]Assuming a two-dimensional cost–performance design space, a design is on the pareto curve, if there is no other design which is better in terms of both cost and performance.

[5]The choice of SPARC was based on the availability of SHADE and a profiling engine; however our approach is clearly applicable to any other (embedded) processor as well.

using figures generated by the Synopsys Design Compiler, and an SRAM cost estimation technique from Catthoor et al. [1998].

We computed the average memory power consumption of each design point, using cache power figures from CACTI [Reinman and Jouppi 1999]. For the main memory power consumption there is a lot of variation between the figures considered by different researchers [Catthoor et al. 1998; Hicks et al. 1997], depending on the main memory type, technology, and bus architecture. The ratio between the energy consumed by on-chip cache accesses and off-chip DRAM accesses varies between one and two orders of magnitude [Hicks et al. 1997]. To keep our technique independent of such technology figures, we allow the designer to input the ratio R as:

$$R = E\_main\_memory\_access / E\_cache\_access$$

where *E_cache_access* is the energy for one cache access, and *E_main_memory_access* is the energy to bring in a full cache line. In our following power computations, we assume a ratio R of 50, relative to the power consumption of an 8k two-way set associative cache with line size of 16 bytes.

In this first set of memory exploration experiments, we consider a simple connectivity architecture, which we refine during the connectivity exploration. The use of multiple memory modules in parallel to service memory access requests from the CPU requires using multiplexers to route the data from these multiple sources. These multiplexers may increase the access time of the memory system, and if this is on the critical path of the clock cycle, it may lead to the increase of the clock cycle. We use access times from CACTI [Reinman and Jouppi 1999] to compute the access time increase and verify that the clock cycle is not affected.

Different cache configurations can be coupled with the memory modules explored, probing different areas of the design space. We present here our technique starting from an instance of such a cache configuration. A more detailed study for different cache configurations can be found in Grun et al. [2001].

We used a set of large real-life multimedia and scientific benchmarks: Compress and Li are from SPEC95, and Vocoder is a GSM voice encoding application.

6.1.2 *Results.* Figure 12 presents the memory design space exploration of the access pattern customizations for the compress application. The compress benchmark exhibits a large variety of access patterns providing many customization opportunities. The x-axis represents the cost (in number of basic gates) and the y-axis represents the overall miss ratio (the miss ratio of the custom memory modules represents the number of accesses where the data is not ready when it is needed by the CPU, divided by the total number of accesses to that module).

The design points marked with a circle represent the memory architectures chosen during the exploration as promising alternatives, and fully simulated for accurate results. The design points marked only with a $\times$ represent the exploration attempts evaluated through fast time-sampling simulation, from which the best cost/gain trade-off is chosen at each exploration

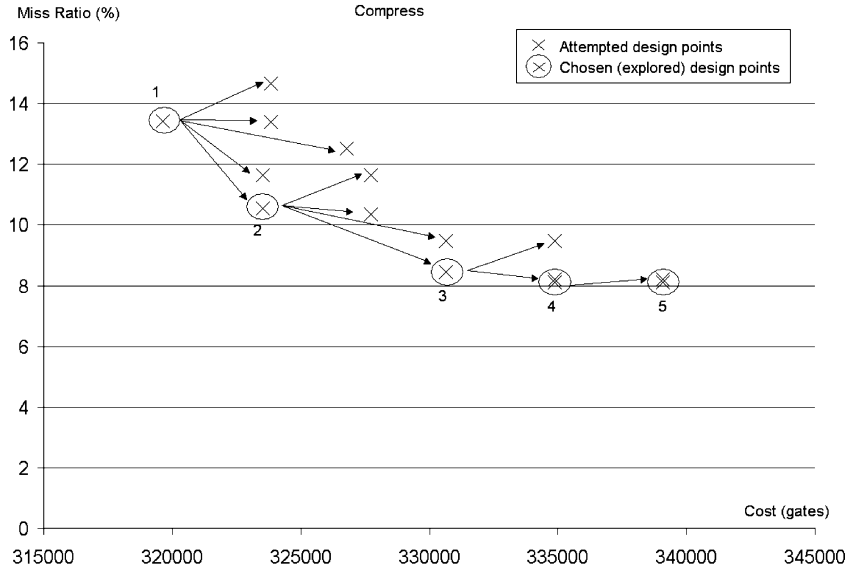

Fig. 12. Miss ratio vs. cost trade-off in memory design space exploration for compress (SPEC95).

step. For each such design we perform full simulation to determine accurate cost/performance/power figures.

The design point labeled 1 represents the initial memory architecture, containing an 8k two-way associative cache. Our exploration algorithm evaluates the first set of customization alternatives, by trying to choose the best access pattern cluster to map to a custom memory module. The best performance gain for the incremental cost is generated by customizing the access pattern cluster containing a reference to the hashing table htab, which uses as an index in the array the access pattern reading the codetab array (the access pattern is htab[codetab[i]]). This new architecture is selected as the next design point in the exploration, labeled 2. After fully simulating the new memory architecture, we continue the exploration by evaluating the further possible customization opportunities, and selecting the best cost/performance ratio. In this way we explore the memory architectures with most promising cost/performance trade-offs, towards the lower bound of the design space.

The miss ratio of the compress application varies between 13.42% for the initial cache-only architecture (for a cost of 319,634 gates), and 8.10% for a memory architecture where three access pattern clusters have been mapped to custom memory modules (for a cost of 334,864 gates). Based on a cost constraint (or alternatively on a performance requirement), the designer can choose the memory architecture that best matches the goals of the system.

To validate our space walking heuristic, and confirm that the chosen design points follow the pareto-curve-like trajectory in the design space, we compared the design points generated by our approach to the full simulation of the design space considering all the memory module allocations and access pattern cluster mappings for the compress example benchmark. Figure 13 shows the design space in terms of the estimated memory design cost (in number of basic

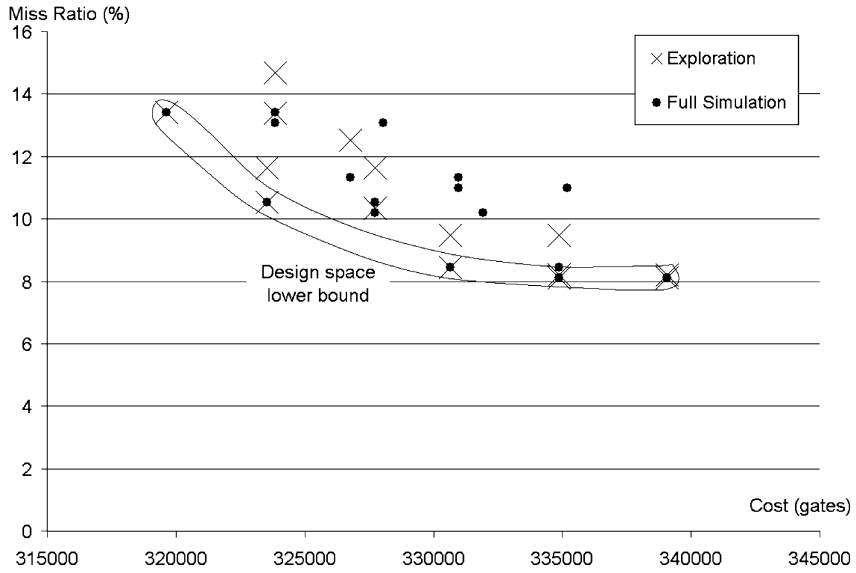

Fig. 13. Exploration heuristic compared to simulation of all access pattern cluster mapping combinations for compress.

gates), and the overall miss rate of the application. The design points marked with a $\times$ represent the points explored by our heuristic. The points marked by a black dot represent a full simulation of all allocation and mapping alternatives. The points on the lower bound of the design space are the most promising, exhibiting the best cost/performance trade-offs. Our algorithm guides the search towards these design points, pruning the noninteresting points in the design space. Our exploration heuristic successfully finds the most promising designs, without fully simulating the whole design space: Each fully simulated design on the lower bound (marked by a black dot) is covered by an explored design (marked by a $\times$).[6] This provides the designer the opportunity to choose the best cost/performance trade-off, without the expense of investigating the whole space.

Table I presents the performance, cost, and power results for a set of large, real-life benchmarks from the multimedia and scientific domains. The first column shows the application, and the second column represents the memory architectures explored for each such benchmark. The third column represents the cost of the memory architecture (in number of basic gates), the fourth column represents the miss ratio for each such design point, the fifth column shows the average memory latency (in cycles), and the last column presents the average memory power consumption, normalized to the initial cache-only architecture (represented by the first design point for each benchmark).

In Table I, we present only the memory architectures with best cost/performance characteristics, chosen during the exploration. The miss ratio

[6]Not all exploration points ($\times$) are covered by a full simulation point (black dot), because some of the exploration points represent estimations only.

Table I. Exploration Results for our Access Pattern based Memory Customization Algorithm

Benchmark	Design Point	Cost (gates)	Miss ratio (%)	Memory Latency (cycles)	Memory Power (normalized)
Compress	1	319,634	13.4200	28.56	1
	2	323,521	10.5400	22.58	1.18
	3	330,657	8.4500	18.42	1.36
	4	334,864	8.1000	17.40	1.41
	5	339,071	8.1000	17.35	1.41
Li	1	319,634	6.9800	15.82	1
	2	323,841	4.6700	11.21	1.23
	3	332,302	4.6200	11.01	1.24
	4	340,763	4.6200	10.96	1.24
Vocoder	1	40,295	1.4600	4.90	1
	2	44,502	1.3600	4.45	1.01
	3	48,709	1.2600	4.16	1.02
	4	53,765	1.2600	4.09	1.03
	5	80,201	0.8100	3.61	0.68
	6	84,408	0.7600	3.26	0.70
	7	88,615	0.7400	3.13	0.70
	8	93,671	0.7400	3.07	0.70

shown in the fourth column represents the number of memory accesses when the data is not yet available in the cache or the custom memory modules when required by the CPU. The average memory latency shown in the fifth column represents the average number of cycles the CPU has to wait for an access to the memory system. Due to the increased hit ratio, and to the fact that the custom memory modules require less latency to access the small FIFO containing the data than the latency required by the large cache tag, data array, and cache control, the average memory latency varies significantly during the exploration.

By customizing the memory architecture based on the access patterns in the application, the memory system performance is significantly improved. For instance, for the compress benchmark, the miss ratio is decreased from 13.4% to 8.10%, representing a 39% miss ratio reduction for a relatively small cost increase. However, this comes at the cost of an increased memory power consumption by a factor between 1.1 and 1.4, mainly because of the increased main memory bandwidth generated by the custom memory modules implementing the access pattern clusters in the application. However, by exploring a varied set of design points, the designer can trade off the cost, power, and performance of the system, to best meet the design goals.

Vocoder is a multmedia benchmark exhibiting mainly stream-like regular access patterns, which behave well with small cache sizes. Since the initial cache of 1k has a small cost of 40,295 gates, there was enough space to double the cache size. The design points 1 through 4 represent the memory architectures containing the 1k cache, whereas the design points 5 through 8 represent the memory architectures containing the 2k cache. As expected, the performance increases significantly when increasing the cost of the memory architecture. However, a surprising result is that the power consumption of the memory

system decreases when using the larger cache: Even though the power consumed by the larger cache accesses increases, the main memory bandwidth decrease due to a lower miss ratio results in a significantly lower main memory power, which translates into a lower memory system power. Clearly, these types of results are difficult to determine by analysis alone, and require a systematic exploration approach to allow the designer to best trade off the different goals of the system.

The wide range of cost, performance, and power trade-offs obtained are due to the aggressive use of the memory access pattern information and customization of the memory architecture beyond the traditional cache architecture.

6.2 Connectivity Exploration

We present here the experimental results taking into account the cost, performance, and power for the full memory system, including both the memory and the connectivity architecture.

6.2.1 *Experimental Setup.* We simulated the design alternatives using our simulator based on the SIMPRESS [Mishra et al. 2001] cycle accurate memory model, and SHADE [Cmelik and Keppel 1993]. We assumed a processor based on the SUN SPARC, and we compiled the applications using gcc. The library of connectivity modules contains information such as the resource usage, latency, pipelining, parallelism, split transaction model, and bitwidth, and the exploration algorithm selects automatically the different connectivity architectures, estimates and prunes the design space, guiding the search towards the most promising designs.

We use a time-sampling [Kessler et al. 1991] estimation to guide the walk through the design space, pruning out the designs that are not interesting. The time-sampling alternates "on-sampling" and "off-sampling" periods, assuming a ratio of 1/9 between the on and off time intervals. We then use full simulation for the most promising designs, to further refine the trade-off choices. The time-sampling estimation does not have a very good absolute accuracy compared to full simulation. However, we use it only for relative incremental decisions to guide the design space search, and the estimation fidelity is sufficient to make good pruning decisions.

6.2.2 *Results.* We performed two sets of experiments: (i) using cost/ performance to drive the memory and connectivity exploration; and (ii) using cost/energy to drive the exploration. In each such set of experiments, we present the effect of the exploration in all the three dimensions (cost, performance, and energy).

(i) Figure 14 shows the cost/performance trade-off for the connectivity exploration of the compress benchmark. The x-axis represents the cost of the memory and connectivity architecture, and the y-axis represents the average memory latency including both the memory and connectivity latencies (e.g., due to the cache misses, bus multiplexing, or bus conflicts).

In this experiment we used cost/ performance to drive the selection algorithms both during the memory modules exploration and during the

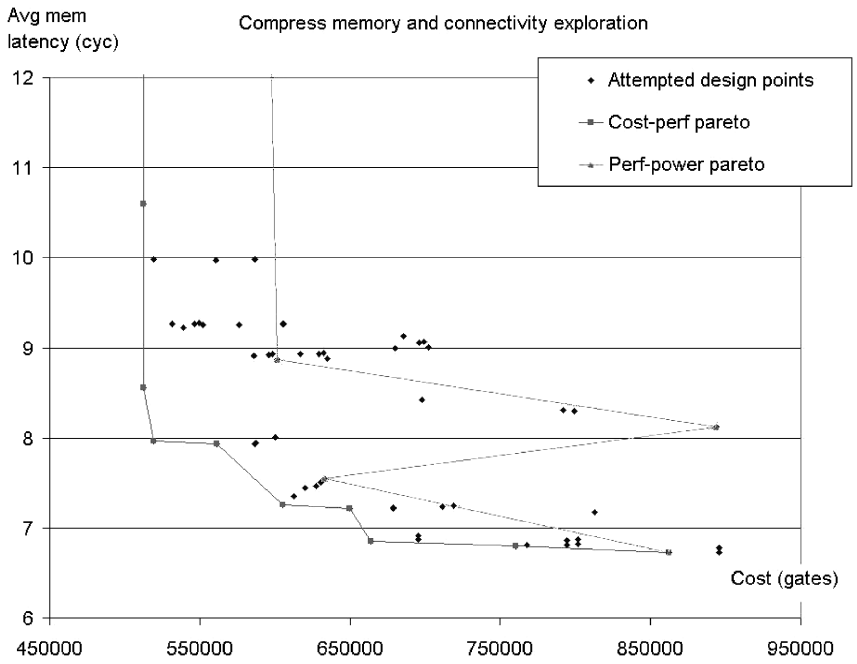

Fig. 14. Cost/performance vs. performance/energy paretos in the cost/performance space for compress.

connectivity exploration. The dots represent the attempted connectivity and memory designs. The line connecting the squares represents the designs on the cost/ performance pareto. However, the designs that have best cost/performance behavior, do not necessarily have good power behavior. The line connecting the triangles represents the designs in the cost/performance space which are on the performance/energy pareto curve.[7] Although the cost/performance and the performance/energy pareto curves do not coincide, they do have a point in common. However, this point has a very large cost. In general, when trading off cost, performance, and energy, the designer has to give up one of the goals in order to optimize the other two. For instance, if the designer wants to optimize performance and energy, typically it will come at the expense of higher cost.

Figure 15 shows the performance/energy trade-offs for the connectivity exploration of the compress benchmark, using cost/performance to drive the selection of the starting memory modules. The x-axis represents the average memory latency, including both the memory and connectivity components. The y-axis represents the average energy per access consumed by the memory and connectivity system. We use energy instead of average power consumption to separate out the impact of the actual energy consumed from the variations in performance. Variations in total latency may give a false indication of the power behavior: For instance, when the performance decreases, the average

[7]The two criteria, cost/performance and power/performance pareto curves are drawn in the same cost/performance design space by taking the cost/performance and performance/power pareto points and drawing their cost and performance coordinates in the cost/performance 2D space.

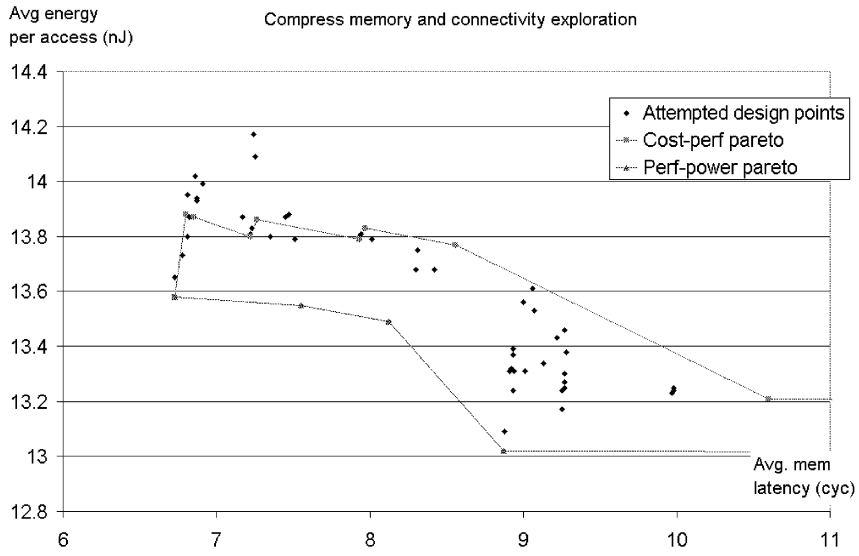

Fig. 15. Cost/performance vs. performance/energy paretos in the performance/energy space for compress.

power may decrease due to the longer latency, but the total energy consumed may be the same.

The line connecting the squares represents the cost/performance pareto points in the performance/energy space. The line connecting the triangles shows the performance/energy pareto points. Again, the best performance/energy points do not necessarily also have low cost. The cost/performance pareto and the performance/energy pareto do not coincide in the performance/energy space. When trading off the three goals of the system, the designer has to give up one of the dimensions in order to optimize the other two. The designs that have good cost and performance behavior (the cost/performance pareto), have in general higher energy consumption (are located on the inside of the performance/energy pareto). The only exception is the common point, which in turn has higher cost.

(ii) Figures 16 and 17 show the cost/performance and performance/energy spaces, respectively, for the exploration results for compress, using cost/energy to drive the memory and connectivity exploration.

In Figure 16, the line connecting the squares represents the cost/performance pareto obtained by the experiments where cost/energy was used to guide the exploration, and the line connecting the stars represents the cost/performance pareto in the case where cost/performance was used throughout the exploration as the driver. As expected, the best cost/performance points obtained during the cost/performance exploration are better in terms of cost and performance than the ones obtained during the cost/energy exploration.

In Figure 17, the line connecting the triangles represents the performance/energy pareto for the cost/performance exploration, whereas the line connecting the stars represents the performance/energy pareto for the cost/energy

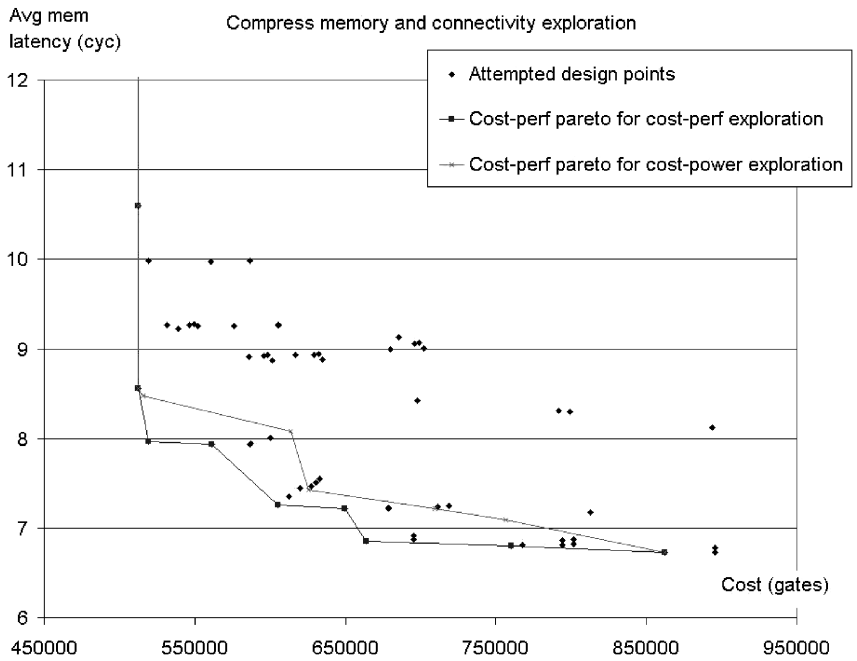

Fig. 16. Cost/performance paretos for the connectivity exploration of compress, assuming cost/performance and cost/energy memory modules exploration.

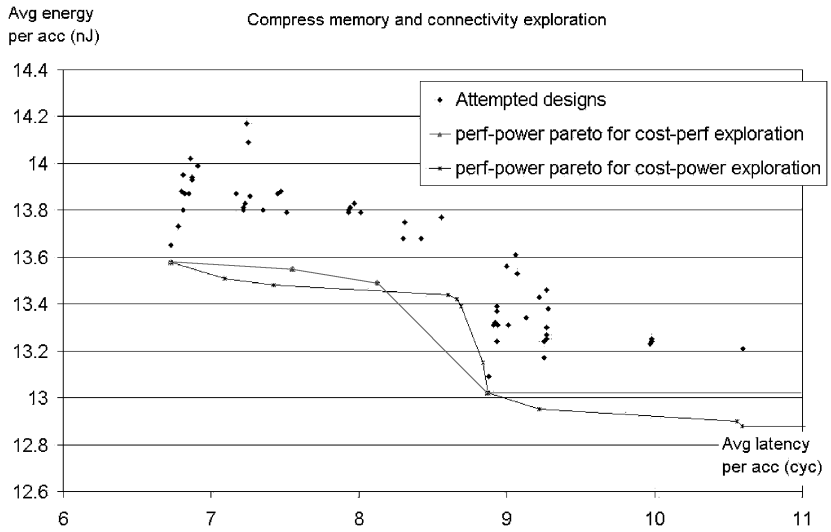

Fig. 17. performance/energy paretos for the connectivity exploration of compress, assuming cost/performance and cost/energy memory modules exploration.

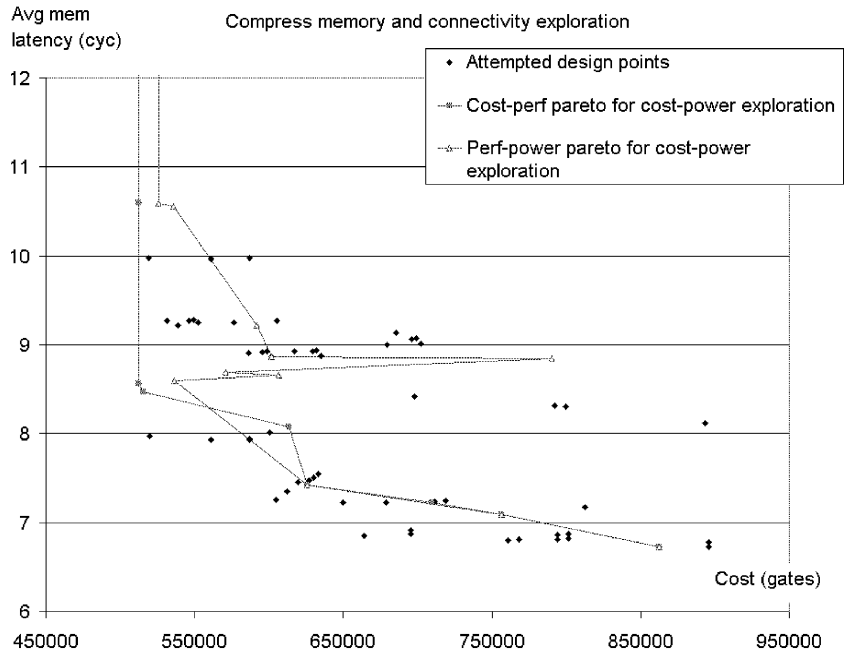

Fig. 18. Cost/performance vs. performance/energy paretos in the cost/performance space for compress, assuming cost–energy memory modules exploration.

exploration. As expected, when using cost/energy to drive the early memory modules exploration (APEX), the overall energy figures are better.

Similarly, the cost/energy space representation of the cost/energy exploration yields better results in terms of cost and energy than the cost/performance exploration.

Figures 18 and 19 show the comparison between the cost/performance and the performance/energy paretos for the connectivity exploration, assuming that the previous phase of memory modules exploration is driven by cost and energy.

Figures 20 and 21 show the comparison between the cost/performance and the performance/energy paretos for the connectivity exploration for vocoder, whereas Figures 22 and 23 show the comparison between the cost/performance and the performance/energy paretos for the connectivity exploration for Li.

Figure 24 shows the analysis of the cost/performance pareto-like points for the compress benchmark. The design points *a* through *k* represent the most promising selected memory–connectivity architectures. Architectures *a* and *b* represent two instances of a traditional cache-only memory configuration, using the AMBA AHB split transaction bus and a dedicated connection. The architectures *c* through *k* represent different instances of novel memory and connectivity architectures, employing SRAMs to store data that is accessed often, DMA-like memory modules to bring in predictable, well-known data structures (such as lists) closer to the CPU, and stream buffers for stream-based accesses. Architecture *c* contains a linked-list DMA-like memory module, implementing a self-indirect data structure, using a MUX-based connection.

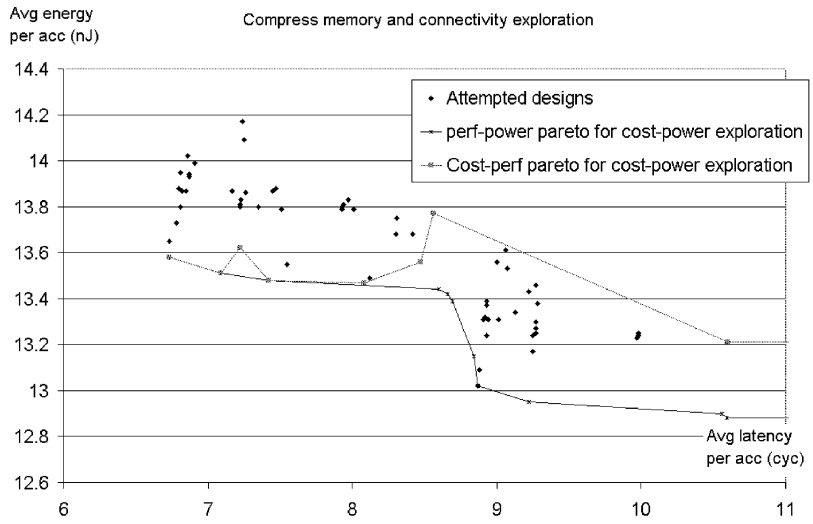

Fig. 19. Cost/performance vs. performance/energy paretos in the performance/energy space for compress, assuming cost–energy memory modules exploration.

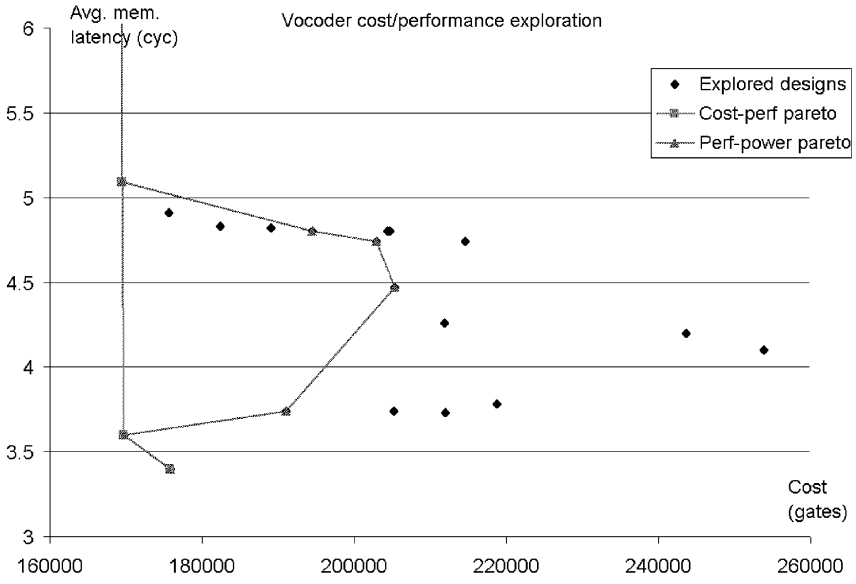

Fig. 20. Cost/performance vs. performance/energy paretos in the cost/performance space for vocoder.

This architecture generates a roughly 10% performance improvement for a small cost increase over the best traditional cache architecture (*b*). The architecture *d* represents the same memory configuration as *c*, but with a connectivity containing both a MUX-based structure and an AMBA APB bus. Similarly, architectures *e* through *k* make use of additional linked-list DMAs, stream buffers, and SRAMs, with MUX-based, AMBA AHB, ASB, and APB

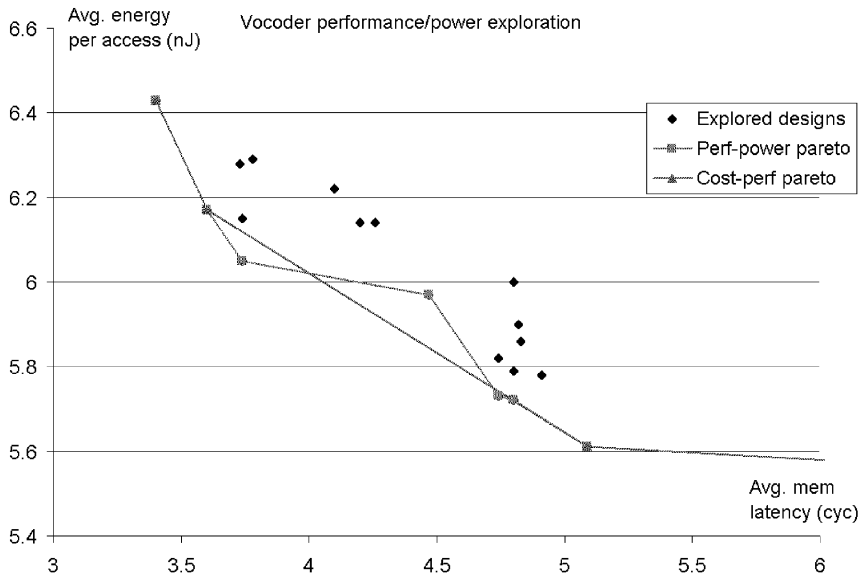

Fig. 21. Cost/performance vs. performance/energy paretos in the performance/energy space for vocoder.

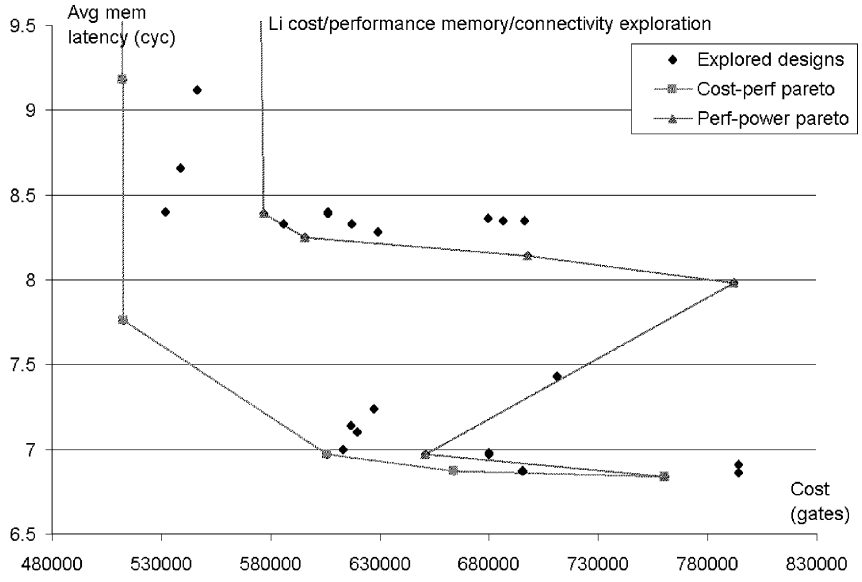

Fig. 22. Cost/performance vs. performance/energy paretos in the cost/performance space for Li.

connections. Architecture *g* generates a roughly 26% performance improvement over the best traditional cache architecture (*b*), for a roughly 30% memory cost increase. Architecture *k* shows the best performance improvement, of roughly 30% over the best traditional cache architecture, for a larger cost increase. Clearly, our memory–connectivity exploration approach generates a significant performance improvement for varied cost configurations, allowing the designer

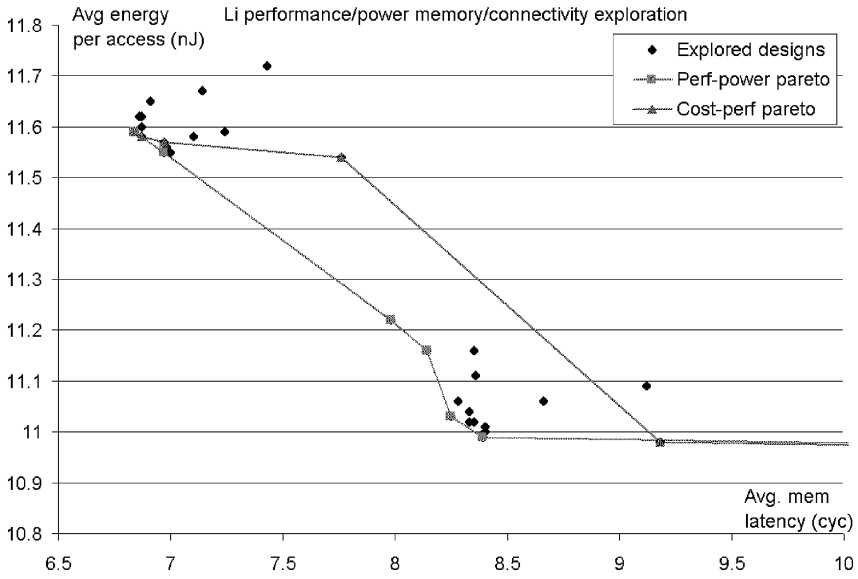

Fig. 23. Cost/performance vs. performance/energy paretos in the performance/energy space for Li.

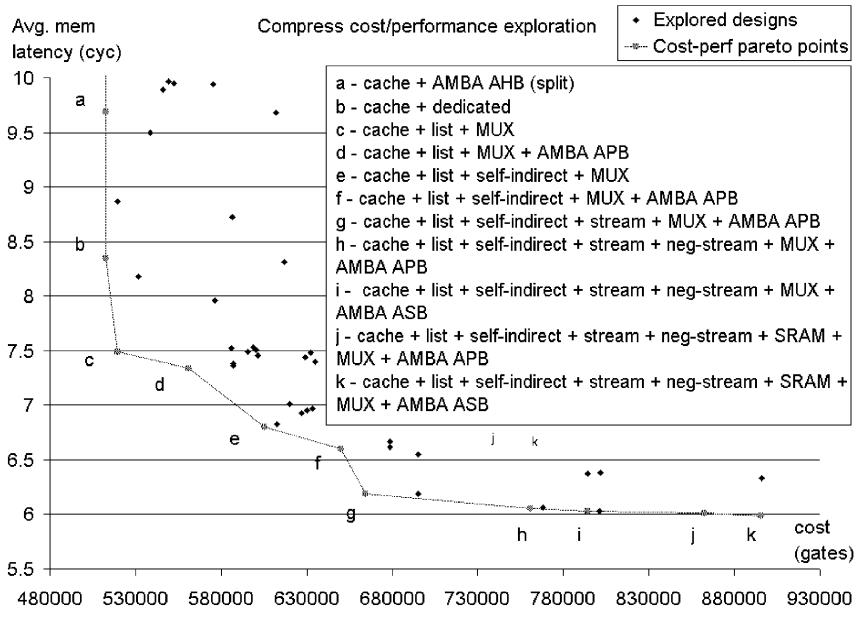

Fig. 24. Analysis of the cost/performance pareto architectures for the compress benchmark.

to select the most promising designs, according to the available chip space and performance requirements.

Figure 25 represents the analysis of the cost/performance pareto-like architectures for the vocoder benchmark. The architectures *a* and *b* represent the traditional cache architectures with AMBA AHB and dedicated connections.

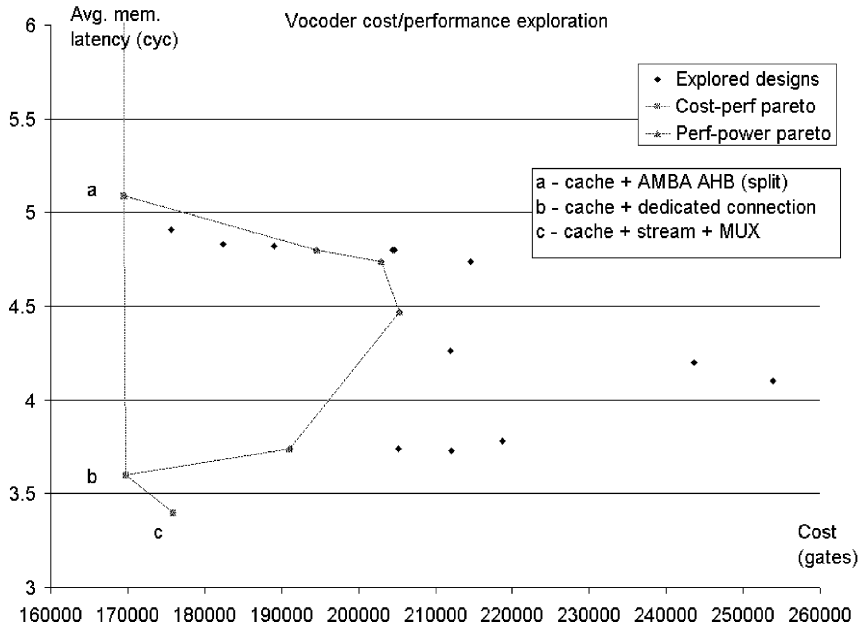

Fig. 25. Analysis of the cost/performance pareto architectures for the vocoder benchmark.

The architecture *c*, containing the traditional cache and a stream buffer, generates a 5% performance improvement over the best traditional cache architecture (*b*) for a roughly 3% cost increase. Due to the fact that the vocoder application is less memory intensive, containing mainly stream-based accesses, which behave well on cache architectures, the performance variation is less significant than in the other benchmarks. However, this illustrates the application-dependent nature of the memory and bandwidth requirements of embedded systems, prompting the need for early memory and connectivity exploration. Clearly, without such an exploration framework it would be difficult to determine through analysis alone the number, amount, and type of memory modules required to match the given performance, energy, and cost criteria.

Figure 26 represents the analysis of the cost/performance pareto-like architectures for the Li benchmark. The memory–connectivity architectures containing novel memory modules, such as linked-list DMAs implementing self-indirect accesses, and stream buffers, connected through AMBA AHB, ASB, and APB buses, generate significant performance variations, allowing the designer to best match the requirements of the system.

In the following, we present the exploration results for the Compress, Li, and Vocoder benchmarks. Due to space limitations, we show only the selected most promising cost/performance designs, in terms of their cost (in basic gates), average memory latency, and average energy consumption per access. In Table II, the first column shows the benchmarks, the second, third, and fourth columns show the cost, average memory latency, and energy consumption for the selected design simulations. The simulation results show significant performance improvement for varied cost and energy characteristics of the designs for all the

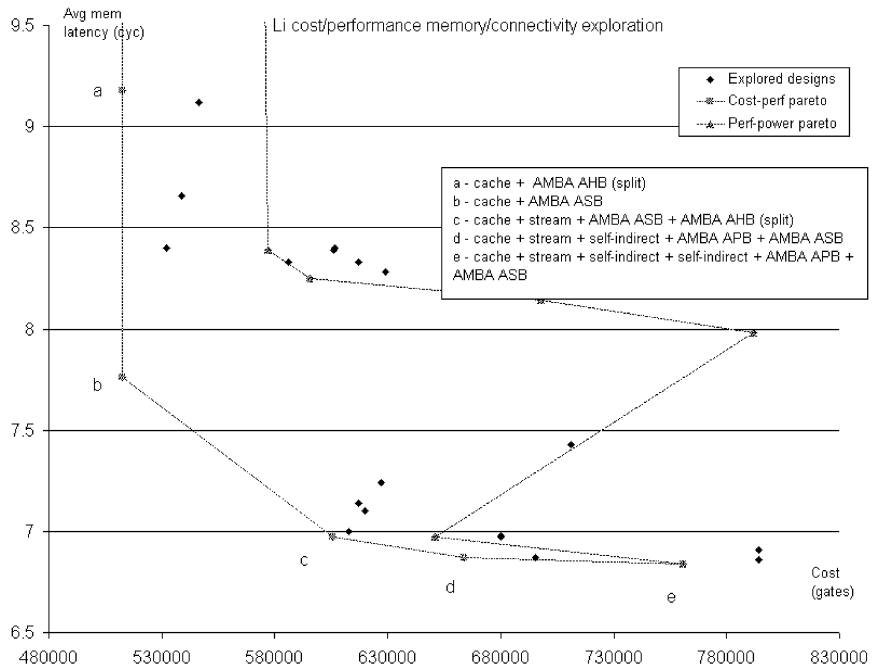

Fig. 26. Analysis of the cost/performance pareto architectures for the Li benchmark.

benchmarks. For instance, when using different memory and connectivity configurations, the performance of the compress and Li benchmarks varies by an order of magnitude. The energy consumption of these benchmarks does not vary significantly, due to the fact that the connectivity consumes a small amount of energy compared to the memory modules.

Table III presents the coverage of the pareto points obtained by our memory modules and connectivity exploration approach. Column 1 shows the benchmark and column 2 shows the category: Time represents the total computation time required for the exploration; coverage shows the percentage of the points on the pareto curve actually found by the exploration. Average distance shows the average percentile deviation in terms of cost, performance, and energy consumption between the pareto points that have not been covered and the closest exploration point that approximates them. Column 3 represents the results for the pruned exploration approach, where only the most promising design points from the memory modules exploration are considered for connectivity space exploration. Column 4 shows the neighborhood exploration results, where the design points in the neighborhood of the selected points are also included in the exploration, and the last column shows the results for the brute-force full space exploration, where all the design points in the exploration space are fully simulated and the pareto curve is fully determined.

The average cost, performance, and energy distance shows the average distance between the points on the pareto curve and the corresponding closest points found by the exploration as the percentile deviation on the corresponding axes. If this average distance is small, it means that even though a design point

Table II. Selected Cost/Performance Designs for the Connectivity Exploration

Benchark	Cost [gates]	Avg mem latency [cycles]	Avg energy [nJ]
Compress	480,775	69.66	13.24
	512,232	62.76	13.52
	512,332	9.69	13.80
	512,532	8.35	14.36
	519,388	7.49	14.44
	561,112	7.34	14.39
	604,941	6.80	14.47
	649,849	6.60	14.39
	664,029	6.19	14.46
	760,543	6.05	14.47
	793,971	6.03	14.54
	862,176	6.01	14.31
	895,604	5.99	14.38
Li	480,775	57.59	10.42
	494,992	57.48	10.43
	512,232	50.29	10.70
	512,332	9.18	10.98
	512,532	7.76	11.54
	605,767	6.97	11.57
	664,029	6.87	11.58
	760,543	6.84	11.59
Vocoder	156,806	16.37	5.05
	169,370	13.28	5.33
	169,481	5.09	5.61
	169,703	3.60	6.17
	175,865	3.40	6.43

Table III. pareto Coverage Results for Our Memory Architecture Exploration Approach

Benchmark	Category	Pruned	Neighborhood	Full
Compress	Time	2 days	2 weeks	1 month
	Coverage [%]	50%	65%	100%
	Avg. cost dist [%]	0.84%	0.59%	0%
	Avg. perf. dist [%]	0.77%	0.60%	0%
	Avg. energ. dist [%]	0.42%	0.28%	0%
Vocoder	Time	24 min	29 min	50 min
	Coverage [%]	83%	100%	100%
	Avg. cost dist [%]	0.29%	0%	0%
	Avg. perf. dist [%]	2.96%	0%	0%
	Avg. energ. dist [%]	0.92%	0%	0%

on the pareto curve has not been found, another design with very close characteristics (cost, performance, energy) is provided (there are no significant gaps in the coverage of the pareto curve).

In the pruned approach, during each design space exploration phase we select for further exploration only the most promising architectures, in the hope that we will find the pareto curve designs without fully simulating the design space. Neighborhood exploration expands the design space explored by also including

the points in the neighborhood of the points selected by the pruned approach. We omitted the Li example from Table III due to the fact that the full simulation computation time was intractable.

The pruned approach significantly reduces the computation time required for the exploration. Moreover, full simulation of the design space is often infeasible (due to prohibitive computation time). Although in general, due to its heuristic nature, the pruned approach may not find all the points on the pareto curve, in practice it finds a large percentage of them, or approximates them well with close alternative designs. For instance, the coverage for the vocoder example shows that 83% of the designs on the pareto curve are successfully found by the pruned exploration. Although the pruned approach does not find all the points on the pareto curve, the average difference between the points on the pareto and the corresponding closest points found by the exploration is 0.29% for cost, 2.96% for performance, and 0.92% for energy. In the compress example the computation time is reduced from 1 month for the full simulation to 2 days, at the expense of less pareto coverage. However, though only 50% of the compress designs are exactly matched by the pruned approach, for every pareto point missed, very close replacements points are generated, resulting in an average distance of 1.95%, 1,83%, and 1.76% in terms of cost, performance, and energy, respectively, to the closest overall pareto point. Thus, our exploration strategy successfully finds most of the design points on the pareto curve without fully simulating the design space. Moreover, even if it misses some of the pareto points, it provides replacement architectures, which approximate well the pareto designs.

The neighborhood exploration explores a wider design space than the pruned approach, providing a better coverage of the pareto curve, at the expense of more computation time. For instance, for the vocoder example, it finds 100% of the pareto points.

6.3 Experiments' Summary

We presented a set of experiments showing the performance, cost, and energy variations generated by our memory and connectivity exploration approach. By extracting and analyzing the different access patterns in the target application, we evaluated different memory and connectivity configurations by allocating specialized memory modules and connectivity components from a library. The pareto curves generated can be used by designers to choose the design points that best meet their goals. Although traditionally designers have relied mainly on intuition and previous experience in making architectural decisions, using such quantitative figures allows more confidence in the decisions, and can lead to improvement in the match between the architecture and target application.

Our experiments have shown that in general when trying to optimize two of the design goals (such as performance and energy), the designer has to give up the third dimension (such us cost). For instance, the performance/energy pareto curves do not coincide with the performance/cost pareto curves.

By performing combined exploration of the memory and connectivity architecture, we obtain a wide range of cost, performance, and energy trade-offs.

Clearly, these types of results are difficult to determine by analysis alone, and require a systematic exploration approach to allow the designer to optimize the trade off among the different goals of the system.

7. SUMMARY

We presented an approach where by analyzing the access patterns in the application we gain valuable insight on the access and storage needs of the input application, and customize the memory architecture to better match these requirements, generating significant performance improvements for varied memory, cost, and power.

Traditionally, designers have attempted to alleviate the memory bottleneck by exploring different cache configurations, with limited use of more special purpose memory modules such as stream buffers. However, though real-life applications contain a large number of memory references to a diverse set of data structures, a significant percentage of all memory accesses in the application are generated from a few instructions that often exhibit well-known, predictable access patterns. This presents a tremendous opportunity to customize the memory architecture to match the needs of the predominant access patterns in the application, and significantly improve the memory system behavior. We present such an approach here, called APEX, that extracts, analyzes, and clusters the most active access patterns in the application and customizes the memory architecture to explore a wide range of cost, performance, and power designs. We generate significant performance improvements for incremental costs and explore a design space beyond the one traditionally considered, allowing the designer to efficiently target the system goals. By intelligently exploring the design space, we guide the search towards the memory architectures with the best cost/performance characteristics and avoid the expensive full simulation of the design space.

Moreover, though the memory modules are important, the connectivity between these modules often has an equally significant impact on the system behavior. We present our connectivity exploration approach (ConEx), which trades off the connectivity performance, power, and cost, using connectivity modules from a library, and allowing the designer to choose the most promising connectivity architectures for the specific design goals. We generate significant performance improvements for incremental costs, and explore a design space beyond the one traditionally considered, allowing the designer to efficiently target the system goals.

We present a set of experiments on large multimedia and scientific examples, where we explored a wide range of cost, performance, and power trade-offs by customizing the memory and connectivity architecture to fit the needs of the access patterns in the applications. Our exploration heuristic found the most promising cost/gain designs compared to the full simulation of the design space, considering all the memory module allocations and access pattern cluster mappings, without the time penalty of investigating the full design space. Future work will address the use of better cost/energy models, as well as evaluating different exploration heuristics.

ACKNOWLEDGMENTS

We would like to acknowledge and thank Ashok Halambi, Prabhat Mishra, Srikanth Srinivasan, Partha Biswas, Aviral Shrivastava, Radu Cornea, and Nick Savoiu, for their contributions to the EXPRESS/ EXPRESSION project.

REFERENCES

ARM AMBA BUS SPECIFICATION. *http://www.arm.com/armwww.ns4/html/AMBA?OpenDocument*.

BAKSHI, S. AND GAJSKI, D. 1995. A memory selection algorithm for high-performance pipelines. In *EURO-DAC*.

CATTHOOR, F., WUYTACK, S., DE GREEF, E., BALASA, F., NACHTERGAELE, L., AND VANDECAPPELLE, A. 1998. *Custom Memory Management Methodology*. Kluwer.

CHEN, H.-M., ZHOU, H., YOUNG, F., WONG, D., YANG, H., AND SHERWANI, N. 1999. Integrated floorplanning and interconnect planning. In *ICCAD*.

CHOU, P., ORTEGA, R., AND BORRIELLO, G. 1995. Interface co-synthesis techniques for embedded systems. In *ICCAD*.

CHUNG, K.-S., GUPTA, R., AND LIU, C. L. 1996. Interface co-synthesis techniques for embedded systems. In *ICCAD*.

CHIUEH, T. C. 1994. Sunder: A programmable hardware prefetch architecture for numerical loops. In *Conference on High Performance Networking and Computing*.

CMELIK, R. AND KEPPEL, D. 1996. Shade: A fast instruction set simulator for execution profiling. Technical report, SUN MICROSYSTEMS.

DAVEAU, J.-M., BEN ISMAIL, T., AND JERRAYA, A. 1995. Synthesis of system-level communication by an allocation-based approach. In *ISSS*.

DENG, Y. AND MALY, W. 2001. Interconnect characteristics of 2.5-d system integration scheme. In *ISPD*.

GAISLER RESEARCH. *www.gaisler.com/leon.html*.

GIVARGIS, T. AND VAHID, F. 1998. Interface exploration for reduced power in core-based systems. In *ISSS*.

GRUN, P., DUTT, N., AND NICOLAU, A. 2000. Memory aware compilation through accurate timing extraction. In *DAC*.

GRUN, P., DUTT, N., AND NICOLAU, A. 2001. Access pattern based local memory customization for low power embedded systems. In *DATE*.

GRUN, P., DUTT, N., AND NICOLAU, A. 2001. APEX: Access pattern based memory architecture exploration. In *ISSS*.

GRUN, P., DUTT, N., AND NICOLAU, A. 2001. Exploring memory architecture through access pattern analysis and clustering. Technical report, #2001-14 University of California, Irvine.

HENNESSY, J. AND PATTERSON, D. 1990. *Computer Architecture: A quantitative approach*. Morgan Kaufmann Publishers Inc, San Mateo, CA.

HICKS, P., WALNOCK, M., AND OWENS, R. M. 1997. Analysis of power consumption in memory hierarchies. In *ISPLED*.

HUMMEL, J., HENDREN, L., AND NICOLAU, A. 1994. A language for conveying the aliasing properties of dynamic, pointer-based data structures. In *Proceedings of the 8th International Parallel Processing Symposium*.

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

KESSLER, R., HILL, M., AND WOOD, D. 1991. A comparison of trace-sampling techniques for multi-megabyte caches. Technical report, University of Wisconsin.

KULKARNI, C. 2001. *Cache optimization for Multimedia Applications*. PhD thesis, IMEC.

LAHIRI, K., RAGHUNATAN, A., LAKSHMINARAYANA, G., AND DEY, S. 2000. Communication architecture tuners: A methodology for hte deisng of high-performance communication architectures for systems-on-chip. In *DAC*.

LIU, D. AND SVENSON, C. 1994. Power consumption estimation in cmos vlsi chips. *IEEE J. of Solid stage Circ. 29*, 6.

MAGUERDICHIAN, S., DRINIC, M., AND KIROVSKI, D. 2001. Latency-driven design of multi-purpose systems-on-chip. In *DAC*.

MISHRA, P., GRUN, P., DUTT, N., AND NICOLAU, A. 2001. Processor-memory co-explotation driven by a memory-aware architecture description language. In *International Conference on VLSI Design*, Bangalore, India.

NARAYAN, S. AND GAJSKI, D. D. 1994. Protocol generation for communication channels. In *DAC*.

PALACHARLA, S. AND KESSLER, R. 1994. Evaluating stream buffers as a secondary cache replacement. In *ISCA*.

PANDA, P., DUTT, N., AND NICOLAU, N. 1999. *Memory Issues in Embedded Systems-on-Chip*. Kluwer.

PARSONS, I., UNRAU, R., SCHAEFFER, J., AND SZAFRON, D. 1997. Pi/ot: Parallel i/o templates. In *Parallel Computing*, *23*, 4–5, (May) 543–570.

PATTERSON, R., GIBSON, G., GINTING, E., STODOLSKY, D., AND ZELENKA, J. 1995. Informed prefeching and caching. In *SIGOPS*.

PRZYBYLSKI, S. 1997. Sorting out the new DRAMs. In *Hot Chips Tutorial*, Stanford, CA.

REINMAN, G. AND JOUPPI, N. 1999. An integrated cache timing and power model. In *Summer Internship Report,* COMPAQ Western Research Lab, Palo-Alto.

SYNOPSYS DESIGN COMPILER. *www.synopsys.com*.

VEIDENBAUM, A., TANG, W., GUPTA, R., NICOLAU, A., AND JI, X. 1999. Adapting cache line size to application behavior. In *ICS*.

VIJAYKRISHNAN, N., KANDEMIR, M., IRWIN, M. J., KIM, H. S., AND YE, W. 2000. Energy-driven integrated hardware-software optimizations using simplepower. In *ISCA*.

WUYTACK, S., CATTHOOR, F., DE JONG, G., LIN, B., AND DE MAN, H. 1996. Flow graph balancing for minimizing the required memory bandwith. In *ISSS*, La Jolla, CA.

Received January 2002; accepted July 2002