APEX: Access Pattern based Memory Architecture Exploration.*

Peter Grun
pgrun@cecs.uci.edu

Nikil Dutt
dutt@cecs.uci.edu

Alex Nicolau
nicolau@cecs.uci.edu

Center for Embedded Computer Systems
University of California, Irvine, CA 92697-3425, USA

ABSTRACT

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, exploring a wide range of cost, performance and power designs. We use a heuristic to prune the design space, guiding the exploration towards the best cost/gain ratios. 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 best target the system goals.

1. INTRODUCTION

In programmable embedded systems, memory represents a major performance and power bottleneck [16]. 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 [9]. However, while 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 in the code. For instance, in Vocoder, a GSM voice coding application with 15K 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.

In the context of disk file systems [14, 15] there have been many approaches to employ the file access pattern to customize the file system to best match the access characteristic of the application. Likewise, many approaches have proposed customizing the processor through special purpose instructions and special functional units to target the predominant computations in embedded applications (such as MAC, FFT, etc.). However, to our knowledge none of the previous approaches have attempted to analyze the memory access patterns information in the application, and use it to aggressively customize the memory architecture. We present here such an approach, where we extract, analyze and cluster the most active memory access patterns in the application, and customize the memory architecture by mixing-and-matching custom memory modules from a 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, allowing the designer to efficiently target the system goals.

In Section 2 we present the related work in the area of memory subsystem optimizations. In Section 3 we present the flow of our approach. In Section 4 we use a large real-life example to illustrate our approach and in Section 5 we present an outline of our Access Pattern based Memory Exploration (APEX) approach. In Section 6 we present a set of experiments that demonstrate the customization of the memory architecture for a set of large multimedia and scientific applications, and present exploration results showing the wide range of performance, power and cost tradeoffs obtained.

2. RELATED WORK

There has been related work in four main domains: (I) Disk file systems and databases, (II) High-level synthesis, (III) Computer Architecture, and (IV) Programmable embedded systems.

(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. [14] present an approach allowing the application programmer to specify

*This work was partially supported by grants from NSF (MIP-9708067), DARPA (F33615-00-C-1632) and a Motorola fellowship.

ISSS'01, October 1-3, 2001, Montréal, Québec, Canada.

the file I/O parallel behavior using a set of templates which can be composed to form more complex access patterns. Patterson et al. [15] 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. [2] 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. [22] present an approach to manage the memory bandwidth by increasing memory port utilization, through memory mapping and code reordering optimizations. Bakshi et al. [1] 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 tradeoffs in the context of programmable embedded systems.

(III) In the domain of Computer Architecture, [9], [12] propose the use of hardware stream buffers to enhance the memory system performance. Reconfigurable cache architectures have been proposed recently [20] 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 while 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. [8] 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 et al. [10], Panda et al. [13] 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. [5] presents 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.

The work we present differs significantly from all the related work in that we aggressively analyze, cluster and map memory access patterns to customized memory architectures; this allows the designer to trade-off performance and power against cost of the memory system.

3. OUR APPROACH

Figure 1 presents the flow of our Access Pattern based Memory Exploration (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 tradeoffs. We prune the design space by using a fast time-sampling simulation to rule-out the non-interesting 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 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.

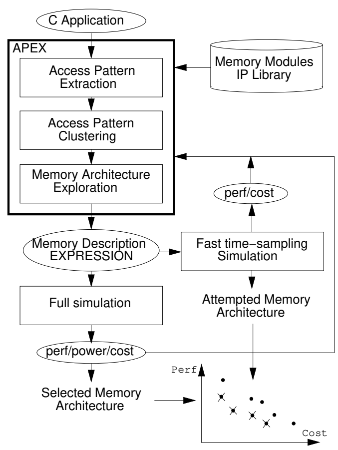

Figure 1: The flow of our Access Pattern based Memory Exploration Approach (APEX).

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 2 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 which 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, while the remaining, less frequent access patterns are serviced by the on-chip cache.

4. ILLUSTRATIVE EXAMPLE

We use the *compress* benchmark (from SPEC95) to illustrate the performance, power and cost trade-offs generated

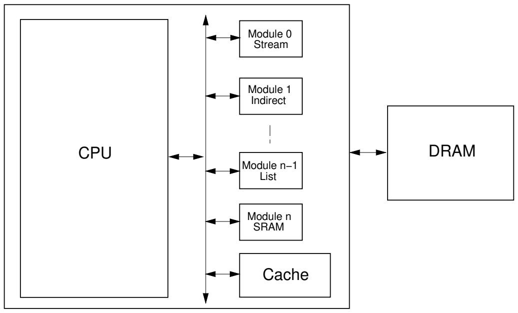

Figure 2: Memory architecture template.

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

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

Figure 3: Example access patterns.

by our approach. The benchmark contains a varied set of access patterns, presenting interesting opportunities for customizing the memory architecture. 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 3 shows an excerpt of code from *compress*, containing references to 3 arrays: htab, codetab, and rmask. htab is a hashing table represented as an array of 69001 unsigned longs (we assume that both longs and ints are stored on 32 bits), codetab is an array of 69001 shorts, and rmask is an array of 9 characters. The sequence of accesses to htab, codetab, and rmask represent 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 patters 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 which are compatible (for instance access patterns which 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 non-unit strides, indirect accesses, self-indirect accesses, linked-list accesses. Although these custom memory modules themselves are not the contribution of the paper (we simply use them as input to our memory architecture exploration algorithm), we briefly describe one such module for illustration purposes. The custom memory modules are based on approaches proposed in the general purpose computing domain [3, 9, 18], 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 self-indirect access pattern, while for the access pattern ap3, due to the small size of the array rmask, we use a small on-chip SRAM [13]. Figure 4 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 which 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. In order to provide the designer with a spectrum of such design points without the time penalty of investigating the full space, we use a heuris-

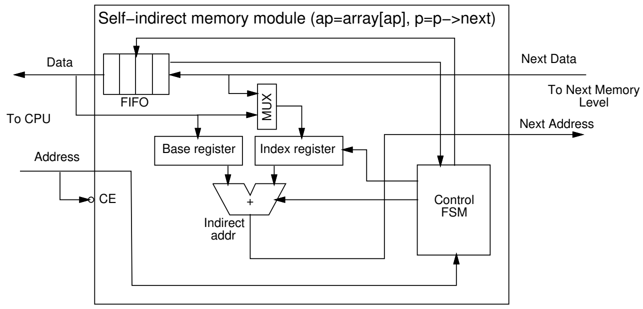

Figure 4: Self-indirect custom memory module.

```
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
```

Figure 5: Access Pattern Clustering algorithm.

tic to select the most promising memory architectures, providing the best cost/performance/power tradeoffs.

For the *compress* benchmark we explore the design space choosing a set of 5 memory architectures which provide advantageous cost/performance tradeoffs. The overall miss rate of the memory system is reduced by 39%, generating a significant performance improvement for varied cost and power characteristics (we present the details of the exploration in Section 6). 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.

5. 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 tradeoff the different goals of the system. It contains two phases: (I) Access pattern clustering and (II) Exploration of custom memory configurations.

5.1 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 5 presents an outline of the access pattern extraction and clustering algorithm. The access pattern clustering algorithm contains 4 steps.

(1) We extract the most active access patterns from the input application. We consider three types of access patterns: (a) Access patterns which 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.

(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 which 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. While 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 [6].

(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.

(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

```
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
```

Figure 6: Exploration algorithm.

are compatible, according to the compatibility criteria determined from the previous step (for a complete description of the compatibility criteria, please refer to [6]). Each such access pattern cluster will be mapped in the following phase to a memory module from the library.

5.2 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 6 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 tradeoff. In order 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. While 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 [6].

6. EXPERIMENTS

We performed a set of experiments on a number of large multimedia and scientific applications to show the performance, cost and power tradeoffs generated by our approach.

6.1 Experimental Setup

We simulated the design alternatives using our simulator based on the SIMPRESS [11] memory model, and SHADE [4]. We assumed a processor based on the SUN SPARC [1], and we compiled the applications using gcc. We estimated the cost of the memory architectures (we assume the cost in equivalent basic gates) using figures generated by the Synopsys Design Compiler [19], and an SRAM cost estimation technique from [2].

We computed the average memory power consumption of each design point, using cache power figures from CACTI [17]. For the main memory power consumption there is a lot of variation between the figures considered by different researchers [2, 7, 21], 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 [7]. In order 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 2-way set associative cache with line size of 16 bytes.

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 [17] to compute the access time increases, 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 [6].

6.2 Results

Figure 7 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

[1]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

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 an X represent the exploration attempts evaluated through fast time-sampling simulation, from which the best cost/gain tradeoff is chosen at each exploration 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 2-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 tradeoffs, towards the lower bound of the design space.

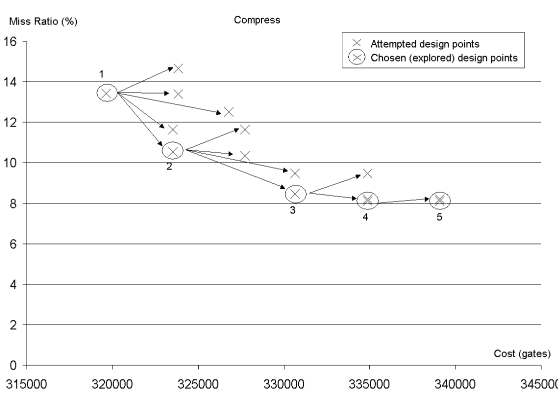

Figure 7: Miss ratio versus cost trade-off in Memory Design Space Exploration for Compress (SPEC95)

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 3 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 which best matches the goals of the system.

In order 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 8 shows the design space in terms of the estimated memory design cost (in number of basic gates), and the overall miss rate of the application. The design points marked with an X 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 tradeoffs. Our algorithm guides the search towards these design points, pruning the non-interesting 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 an X) [2]. This provides the designer the opportunity to choose the best cost/performance trade-off, without the expense of investigating the whole space.

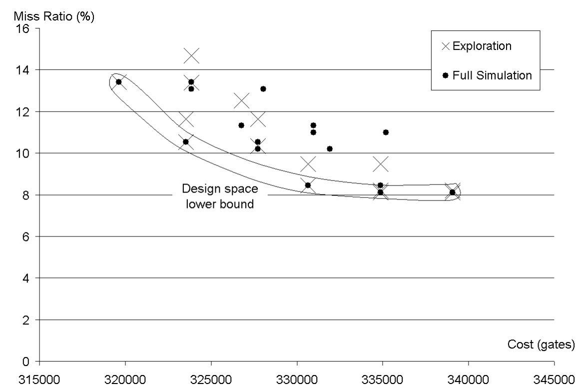

Figure 8: Exploration heuristic compared to simulation of all access pattern cluster mapping combinations for Compress

Table 1 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 1 we present only the memory architectures with best cost/performance characteristics, chosen during the exploration. The miss ratio 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 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

[2]Not all exploration points (X) are covered by a full simulation point (black dot), since some of the exploration points represent estimations only

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 due to 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 tradeoff the cost, power and performance of the system, to best meet the design goals.

Benchmark	Design Point	Cost (gates)	Miss ratio (%)	Mem Latency (cycles)	Mem. Power (normalized)
Compress	1	319634	13.4200	28.56	1
	2	323521	10.5400	22.58	1.18
	3	330657	8.4500	18.42	1.36
	4	334864	8.1000	17.40	1.41
	5	339071	8.1000	17.35	1.41
li	1	319634	6.9800	15.82	1
	2	323841	4.6700	11.21	1.23
	3	332302	4.6200	11.01	1.24
	4	340763	4.6200	10.96	1.24
vocoder	1	40295	1.4600	4.90	1
	2	44502	1.3600	4.45	1.01
	3	48709	1.2600	4.16	1.02
	4	53765	1.2600	4.09	1.03
	5	80201	0.8100	3.61	0.68
	6	84408	0.7600	3.26	0.70
	7	88615	0.7400	3.13	0.70
	8	93671	0.7400	3.07	0.70

Table 1: Exploration results for our Access Pattern based Memory Customization algorithm.

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, while 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, this type 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 tradeoffs obtained are due to the aggressive use of the memory access pattern information, and customization of the memory architecture beyond the traditional cache architecture.

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 [9]. However, while 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, which 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 presented here such an approach 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.

We presented a set of experiments on large multimedia and scientific examples, where we explored a wide range of cost, performance and power tradeoffs, by customizing the memory 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.

In this work we assumed a simple connectivity model between the memory modules and the CPU. In our future work we plan to explore the connectivity space, by using specific on-chip and off-chip busses and connections from a bus IP library.

8. 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.

9. REFERENCES

[1] S. Bakshi and D. Gajski. A memory selection algorithm for high-performance pipelines. In *EURO-DAC*, 1995.

[2] F. Catthoor, S. Wuytack, E. De Greef, F. Balasa, L. Nachtergaele, and A. Vandecappelle. *Custom Memory Management Methodology*. Kluwer, 1998.

[3] Tzi cker Chiueh. Sunder: A programmable hardware prefetch architecture for numerical loops. In *Conference on High Performance Networking and Computing*, 1994.

[4] R. Cmelik and D. Keppel. Shade: A fast instruction set simulator for execution profiling. Technical report, SUN MICROSYSTEMS, 1993.

[5] P. Grun, N. Dutt, and A. Nicolau. Access pattern based local memory customization for low power embedded systems. In *DATE*, 2001.

[6] P. Grun, N. Dutt, and A. Nicolau. Exploring memory architecture through access pattern analysis and clustering. Technical report, University of California, Irvine, 2001.

[7] P. Hicks, M. Walnock, and R.M. Owens. Analysis of power consumption in memory hierarchies. In *ISPLED*, 1997.

[8] J. Hummel, L Hendren, and A Nicolau. A language for conveying the aliasing properties of dynamic, pointer-based data structures. In *Proceedings of the 8th International Parallel Processing Symposium*, 1994.

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

[10] C. Kulkarni. *Cache optimization for Multimedia Applications*. PhD thesis, IMEC, 2001.

[11] P. Mishra, P. Grun, N. Dutt, and A. Nicolau. Processor-memory co-explotation driven by a memory-aware architecture description language. In *International Conference on VLSI Design*, Bangalore, India, 2001.

[12] S. Palacharla and R. Kessler. Evaluating stream buffers as a secondary cache replacement. In *ISCA*, 1994.

[13] P. Panda, N. Dutt, and N. Nicolau. *Memory Issues in Embedded Systems-on-Chip*. Kluwer, 1999.

[14] I. Parsons, R. Unrau, J. Schaeffer, and D. Szafron. Pi/ot: Parallel i/o templates. In *Parallel Computing, Vol. 23, No. 4-5, pp. 543-570, May 1997.*

[15] R. Patterson, G. Gibson, E. Ginting, D. Stodolsky, and J. Zelenka. Informed prefeching and caching. In *SIGOPS*, 1995.

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

[17] G. Reinman and N. Jouppi. An integrated cache timing and power model. In *Summer Internship Report, COMPAQ Western Research Lab, Palo-Alto*, 1999.

[18] A. Roth, A. Moshovos, and G. Sohi. Dependence based prefetching for linked data structures. In *ASPLOS*, 1998.

[19] Synopsys Design Compiler. *www.synopsys.com*.

[20] A. Veidenbaum, W. Tang, R. Gupta, A. Nicolau, and X. Ji. Adapting cache line size to application behavior. In *ICS*, 1999.

[21] N. Vijaykrishnan, M. Kandemir, M. J. Irwin, H. S. Kim, and W. Ye. Energy-driven integrated hardware-software optimizations using simplepower. In *ISCA*, 2000.

[22] S. Wuytack, F. Catthoor, G. de Jong, B. Lin, and H. De Man. Flow graph balancing for minimizing the required memory bandwith. In *ISSS*, La Jolla, CA, 1996.