# Access Pattern based Local Memory Customization for Low Power Embedded Systems.*

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, the local memory relied on a large cache to store all the variables in the application. However, especially in large real-life applications, different types of data exhibit divergent types of locality and access patterns, with diverse locality and bandwidth needs. Traditional caches had to compromise between the different types of locality required by the access patterns, and trade-off performance against bandwidth requirement. Instead, our approach customizes the local memory architecture matching the diverse access patterns and locality types present in the application, to reduce the main memory bandwidth requirement, and significantly improve power consumption, without sacrificing performance. Our approach generated an average 30% memory power reduction without degrading performance on a set of large multimedia/general purpose applications and scientific kernels, over the best traditional cache configuration of similar size, demonstrating the utility of our algorithm.*

## 1 Introduction

In recent embedded systems architecture, memory represents a major performance and power bottleneck [25]. Increasingly, the performance gap between processor and memory has been addressed using faster memory modules (such as SDRAM, RAMBUS, DDRAM [7, 10]), or by fetching more data into local memories (for instance through prefetching [3, 5, 19]). However, such techniques while aggressively targeting performance, rely on significantly increased bandwidth. For instance, the burst-mode, page-mode, pipelined accesses [24, 25] in SDRAM, RAMBUS, increase performance by increasing the memory bandwidth. Similarly, prefetching results in substantial bandwidth increase, due to un-avoidable useless/redundant fetches [21]. These additional bandwidth requirements result in significant rise in power consumption. Instead, while improving performance, we would also like to reduce power by modulating the bandwidth. We present here such an approach, where we reduce the bandwidth and power consumption, without sacrificing performance. The key idea is that by customizing the local memory architecture for the specific access patterns in the application, it is possible to achieve comparable or better performance, while decreasing the main memory bandwidth thus generating significant power savings.

Whereas traditionally local memory optimizations have concentrated on increasing hit ratio, or reducing transition activity through address coding or program transformations, to our knowledge, no prior work has addressed reducing the main memory bandwidth and power consumption by customizing the local memory modules to the specific application access patterns and locality types.

Traditionally, a simple cache hierarchy, containing a level one and possibly level two cache was used, and assuming that by increasing the hit ratio, also the number of accesses to the main memory will decrease. However, this is not always true, since increasing the hit ratio past a point comes at the cost of bringing substantially more data into the levels closer to the CPU (e.g, by increasing the cache line size, or through prefetching [21]), resulting in useless or redundant fetches, and increased memory traffic.

Moreover, there is a large variation in the access patterns and locality type of the data not only between different domains [6, 18, 26], but also within the the same domain and application [8]. For instance, while scalars (such as counters) generally exhibit large temporal locality, and moderate spatial locality, vectors with large stride exhibit no spatial locality, and vectors with small stride exhibit large spatial locality, and may or may not exhibit temporal locality.

By customizing the local memory modules according to the prevalent access modes in the application, it is possible to fine-tune the local memory management mechanism for the different types of locality, resulting in a more judicious main memory bandwidth management, and substantial power savings. In this paper we present such an approach, where we use different types of local memory modules to store variables with high temporal and high spatial locality, to improve the utilization of the main memory bandwidth, and generate power reduction, without sacrificing performance.

In Section 2 we compare our approach to previous work in the area. In Section 3 we present our approach. In Section 4 we use an example application to illustrate our technique, and then present our exploration algorithm in Section 5. In Section 6 we present a set of experiments on multimedia, general

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

purpose and scientific benchmarks, showing the bandwidth and power reduction obtained by our approach. We conclude with a short summary in Section 7.

## 2 Related Work

Related work on memory optimizations has addressed four main areas: cache hit ratio optimizations, memory bandwidth management optimizations, memory subsystem power optimizations and cache architecture optimizations.

An extensive amount of work has concentrated on cache hit ratio optimizations through code transformations (such as loop interchange, loop blocking) [22, 30], and memory mapping optimizations (such as padding, tiling) [4, 22] to improve locality. While such techniques can improve the different types of locality, they cannot substantially change the access patterns characteristics (for instance they cannot change a stream access or a random access into a highly temporal access). We capitalize on the large variation of access patterns and locality types in the application, by customizing the local memory modules to meet these locality needs. Moreover, while previous techniques aggressively targeted performance improvement by improving the hit ratio, we complement this work by addressing memory bandwidth and power reduction.

Wuytack et. al. [31] present an approach to manage the memory bandwidth by increasing memory port utilization, through memory mapping and code reordering optimizations. Our technique complements this work by improving the bandwidth utilization in the presence of caches.

In [11] we presented the Miss Traffic Management (MIST) compiler optimization technique which explicitly manages the main memory traffic, to hide the latency of the cache misses, and generate performance improvements. Here, we explore different local memory architectures, targeting power reduction through local memory customization. Once the memory bandwidth has been reduced, the MIST algorithm can be applied at the later compilation stage, to further improve performance by overlapping the remaining main memory traffic with other CPU and cache hit operations. In [10] we proposed a compiler technique to improve the memory bandwidth by exploiting special features of SDRAM, RAMBUS, DDRAM modules, such as page-mode, burst-mode, pipelined accesses [24]. Rixner et. al [7] proposed a hardware memory-controller solution to this problem. However, both these approaches in effect improve performance by increasing the bandwidth provided by the memory modules. Instead, our goal in this work is to reduce the memory bandwidth requirement of the application.

Compiler transformations addressing memory system power reduction have concentrated mainly on address coding to reduce the transition activity on the address bus [23]. Benini et. al [1] presented an encoding/decoding interface logic synthesis approach that minimizes the average number of transitions on global buses. Instead, we reduce the number of accesses to the main memory through customization of the local memory architecture. Kulkarni et. al [17] presented a combined in-place memory mapping and code transformations approach to reduce the memory system power for different cache sizes. Instead of this traditional cache architecture, we use a more flexible local memory, containing possibly multiple modules targeting different types of locality, to fine-tune the balance between performance and power.

Reconfigurable cache architectures have been proposed recently [29] 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 results in a power overhead which is prohibitive in embedded systems. Instead, we statically target the local memory architecture to the data access patterns, and significantly reduce power. Adve et. al [27] propose a reconfigurable cache, which allows using the available on-chip SRAM space in CPU-bound applications for different hardware supported optimizations (such as hardware lookup tables for instruction reuse or hardware prefetching).

The work closest to ours is by Gonzales et. al [8], which presents a double cache architecture, containing a temporal cache and a spatial cache, targeting general purpose processors, and relying on a dynamic prediction mechanism to route the data to either the spatial or the temporal caches, based on a history buffer. Instead, in our approach we allocate the variables statically to the different local memory modules, avoiding the power and area overhead of the dynamic prediction mechanism. Moreover, while in their approach they target performance, our goal is to reduce power consumption, while keeping the performance comparable or better.

## 3 Our approach

Different types of data usually exhibit different types of locality properties and access patterns. For instance, scalar variables tend to have high temporal locality (e.g., counters, indices), with moderate degree of spatial locality, since variables stored close to each other in memory are not always referenced at the same time. Vectors with large stride have poor spatial locality, may or may not have temporal locality, and may be suitable for stream access. Vectors with small strides exhibit large spatial locality, while random accesses have in general poor locality.

In the traditional single-cache hierarchy architecture, all variables in an application were cached by a single cache. However, especially in large real-life programs, with many variables exhibiting such varied access patterns and locality characteristics, it is very difficult to find a single one-size-fits-all cache, which efficiently exploits all locality types. For instance, when increasing the cache line size to improve spatial locality, many useless fetches from the main memory are introduced, in the variables with little or no spatial locality.

In this paper we present a technique to customize the local memory architecture for the different access and locality patterns. We first cluster the variables according to different

types of locality, then match the needs of each such cluster using different local memory modules. For instance, by using temporal caches for variables with high temporal locality, and spatial caches for variables with high spatial locality, we reduce the main memory bandwidth requirement, and significantly reduce power consumption. Temporal caches are caches with small line size (of a few bytes), and spatial caches exhibit large line sizes to exploit the spatial locality.

Figure 1 presents the flow of our approach. Our local memory customization technique is part of the MemorEx early Design Space Exploration (DSE) approach. The design starts by selecting a set of local memory modules from a local memory IP library, along with main memory modules from a main memory IP library and processors from a processor IP library. Our EXPRESSION Architectural Description Language (ADL) [13], is used to capture the processor-memory subsystem [20], and retarget the profiler for early memory subsystem profiling, as well as the latter compilation [10, 11, 12] and simulation stages [16]. The PaLM algorithm customizes the local memory architecture, according to the locality types of the variables present in the input C application.

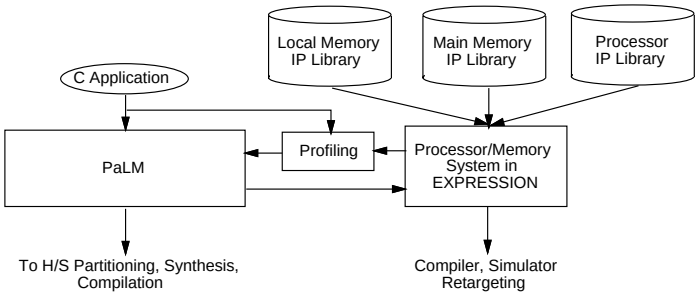


**Figure 1. The flow of our approach**

## 4 Illustrative example

We use the Go benchmark from SPECINT95 to illustrate the power savings generated by our approach. The Go program is an application for playing Go, exhibiting a varied set of data access patterns and locality types. The characteristics of the program are presented in Table 1.

We present the system performance and power in 2 cases: (I) the traditional local memory architecture, containing a single cache, tuned to best balance between temporal and spatial locality for the specific mixture of access patterns in the application, and (II) our customized local memory architecture, containing a temporal and a spatial cache to specifically exploit the types of locality of the different variables.

Using simulation results from the WATTCH power simulator [2] for an 8KB 2-way set associative on-chip cache in 0.35 micron technology, the cache accesses consume an average of 0.025W and idle cycles consume 0.0025W (at a frequency of 100MHz). The off-chip main memory dissipates an average of 2W for accessing 8 bits at 100MHz [15]. Using the bus power estimation function from [4], the main memory interconnect dissipates 0.6W for accesses at 100MHz. We assume the total local memory space available is 8KB.

(I) In the traditional cache architecture, a single cache module services all the variables in the application. In order to determine the best such traditional cache configuration, which best compromises the different types of locality present in the application, we explore the hit ratio and memory bandwidth requirements for different cache line sizes for our example.

Figure 2 presents the hit ratio and bandwidth variation with the cache line size for the go benchmark, assuming an 8KB 2-way set associative cache, and varying the cache line size. The X axis represents the cache line sizes in 4 byte words, and the Y axis represents the cache hit ratio and main memory bandwidth, computed as the average number of bytes requested from the main memory per cycle.

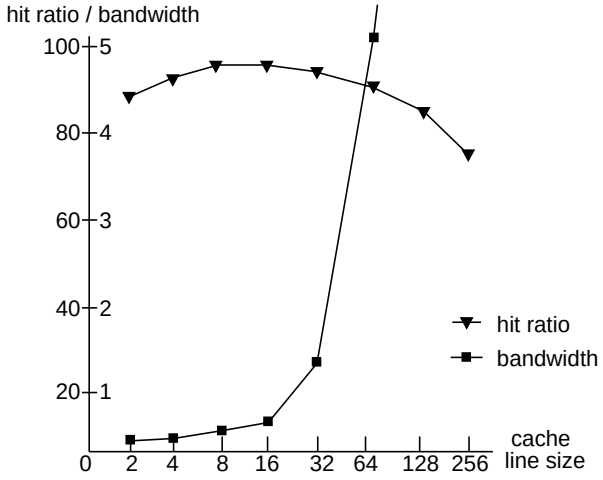


**Figure 2. The cache hit ratio and memory bandwidth variation with the cache line size for the go benchmark**

| Bench | # lines of code | # scalars | # arrays | array sizes | locality types |
|---|---|---|---|---|---|
| go | 30K | 86 | 202 | 8 - 54380 [bytes] | temporal, spatial, stream, random |

**Table 1. Characteristics of Go benchmark.**

When increasing the cache line size, the hit ratio initially increases, reaching a peak of 95% for line sizes of 8 and 16 words, then decreases for larger cache lines. On the other hand, the bandwidth increases with the cache line size slowly initially, then explodes for line sizes larger than 32 words. The best balance between hit ratio and bandwidth is reached for line sizes 8 and 16: hit ratios are both 95%, while the bandwidth for line size 8 is 0.32 bytes/cycle, significantly lower than for line size 16 (0.58 bytes/cycle). Since the power consumption is directly influenced by the main memory accesses, the line size 8 cache represents the best configuration.

The average main memory power consumption for the traditional approach is the average number of main memory ac-

cesses per cycle multiplied with the power consumed by an access:

$$\begin{aligned} Trad\_power &= bandwidth * (dram\_power + bus\_power) \\ &= 0.32bytes/cycle * (2W + 0.6W) = 0.832W \end{aligned}$$

In the following we will show how by customizing the local memory architecture, it is possible to significantly reduce the power consumption of the memory system, without sacrificing performance.

(II) In our approach, we use two local memory modules to separately match the needs of variables with high temporal locality and variables with high spatial locality from the example application. Instead of the single cache module from the traditional approach, we use two smaller caches: a temporal cache, exhibiting small line size, to store the variables with small spatial locality, and a spatial cache, with larger line size, to store the variables with high spatial locality. By analyzing the access patterns in our example application, and clustering the variables into groups exhibiting different locality patterns, we determine the amount of local memory needed for each such type of locality.

In the Go benchmark, the scalars and small arrays have mostly high temporal locality, and moderate spatial locality. For instance, ldir, an array of 52 integers storing a table of directions, has large temporal locality (an average of 255 accesses per value), and small spatial locality (only 28% of the neighboring values are reused). On the other hand, the large arrays, tend to have large spatial and small to moderate temporal locality. For instance, the array armyrun of 3600 integers, storing the number of liberties around an army, averages 1 access per value, and 100% of the neighboring values are reused. Due to the distribution of the temporal and spatial locality among variables, we use a small temporal cache, to hold mostly scalars and small arrays, and some of the larger arrays with small spatial locality, and a larger spatial cache, exhibiting a large line size, to store the large arrays with high spatial locality. For our example, the temporal cache has a line size of 4 words, and a size of 2KB, and the spatial cache has a line size of 8 words, and a size of 6KB (we assume words of 4 bytes).

The hit ratio for the customized architecture is 95%, similar to the best traditional cache configuration (in fact marginally better than the line size 8 traditional cache configuration). Due to the better match between the cache line sizes and the types of locality present in the application, a large number of useless main memory accesses are avoided, and the main memory bandwidth is reduced to 0.26 bytes/cycle, representing a 23% reduction over the traditional cache configuration.

There are 3 components which influence the power consumption in our approach. (a) the main memory system power consumption in our architecture is reduced by the smaller number of main memory accesses, (b) the power consumed by each cache access is reduced due to the smaller size of the caches (since the cache power grows with the cache size, and the total number of cache accesses is the same as in the traditional approach), and (c) the cache idle power is increased due to the extra cache control.

(a) The main memory power consumption for our approach is equal to the average number of main memory accesses (the main memory bandwidth) multiplied with the memory access power consumption:

$$\begin{aligned} PaLM\_mem\_power &= bandwidth * (dram\_power + bus\_power) \\ &= 0.26bytes/cycle * 2.6W = 0.67W \end{aligned}$$

(b) The power consumed by a cache during an access grows with the size of the cache. Since in our approach each of the two caches have smaller size than the cache in the traditional approach (the two caches in our approach use the same space as the single cache in the traditional approach), each cache access will consume less power than the traditional large cache, while the total number of cache accesses remains the same. However, for simplicity of the explanations we do not include this power saving in our computations.

(c) During cache idle cycles, instead of the traditional single cache, in our approach two caches consume idle power. However, since the main memory accesses are much more power hungry than the cache control, this overhead is small compared to the savings obtained by reducing the main memory bandwidth. The overhead due to the extra idle power consumed by the additional cache in this example is:

$$cache\_ctrl\_overhead = 0.0025W$$

Therefore the power consumed in our approach, including the cache control power overhead is:

$$\begin{aligned} PaLM\_power &= PaLM\_mem\_power + cache\_ctrl\_overhead \\ &= 0.6725W \end{aligned}$$

The total power saving between our approach and the traditional cache, considering the overhead due to the extra cache control is:

$$\begin{aligned} PaLM\_power\_saving &= Trad\_power - PaLM\_power \\ &= 0.16W \end{aligned}$$

The power reduction for our approach represents a 23% saving over the best traditional configuration, while the hit ratio is unchanged (in fact marginally better). Thus, by customizing the local memory architecture to better fit the varied types of locality in the application, it is possible to manage more judiciously the memory bandwidth, and generate significant power savings without sacrificing performance.

# 5 Pattern Based Local Memory Customization Algorithm

We present in the following our Pattern Based Local Memory Customization for Low Power (PaLM) algorithm, which customizes the local memory architecture to match the locality needs of the different access patterns in the application, and generate substantial power savings. The PaLM algorithm receives as input the C application, along with the available local memory space, and generates as output the customized local memory architecture along with the mapping of the variables to different local memory modules.

**Algorithm:** Access Pattern based Local Memory Customization for Low Power
**Input:** The C Application and the amount of available local memory,
**Output:** Local Memory Organization and variables mapping
**Begin PaLM**
1. Compute the access patterns and locality types in the application.
2. Cluster the variables according to the locality types.
3. Choose the Local Memory Architecture Template.
4. Selecte the Local Memory Modules Characteristics.
5. Map the variables to the Local Memory Modules.

**End PaLM**

**Figure 3. The Access Pattern based Local Memory Customization (PaLM) Algorithm.**

Figure 3 presents the Local Memory Customization algorithm, composed of 5 steps. The first step computes the access patterns and locality types in the input application. The second step clusters the variables according to the similarity in the access patterns. The third step chooses the local memory architecture template, while the fourth step chooses the characteristics of each local memory module. The last step performs mapping of the variables to specific local memory modules.

The first step extracts information about the access patterns and locality types for the different variables in the application. Along with variable sizes, this information will be used to customize the local memory architecture, and map the variables to specific local memory modules. For each variable we determine a set of metrics which characterize the locality type of the variable, and suitability to different caching approaches. The metrics we determine at this level are symbolic, without considering physical characteristics of the local memory modules. The later steps of the algorithm will introduce more physical characteristics of the memory organization, and will refine the decisions based on profiling.

In order to characterize suitability for temporal locality, for each variable (scalar or array) we determine the average number of accesses per value. as well as the average reuse distance. The reuse distance [14] represents the distance between two consecutive accesses to a particular value. The average reuse distance represents the average over all reuses of a value, and over all values in the variable (an array may be composed of multiple values).

$$avg\_accesses\_per\_value(var) = \frac{no\_of\_accesses(var)}{no\_of\_values\_accessed(var)}$$

$$avg\_reuse\_distance(var) = \frac{SUM(reuse\_distance(var))}{no\_of\_reuses(var)}$$

To characterize spatial locality, we determine the percentage of neighboring values which are accessed in the future. We define the term neighboring by using a spatial locality function F, and a block size $B_s$, where F(addr) represents the set of values in the same spatial locality block as addr, and $B_s$ represents the spatial locality block size. For instance, in the traditional cache, the blocks translate into cache lines, and the locality function represents the cache line holding the particular address. Spatial density of the accesses represents the average number of values accessed in a block, over the block size (we consider only the blocks where at least one value is accessed).

$$spatial\_density(var) = \frac{avg\_no\_values\_accessed(block, var)}{B_s}$$

The second step of the PaLM algorithm clusters the variables according to the locality metrics computed in the previous step. We use a set of heuristics, to categorize the variables according to their spatial and temporal locality. Variables with high number of accesses per value and low reuse distance, are good candidates to store in a temporal cache. Similarly, values with moderate accesses per value and low reuse distance run a high chance to benefit from temporal locality. On the other hand, variables with high reuse distance, are likely to be thrown out of the cache before being reused. Variables with low accesses per value, profit less from temporal locality.

The spatial density is a number between 0 and 1. Variables with a large spatial density are likely to profit from spatial locality caches. On the other hand variables with low spatial density, when mapped to spatial caches will result in useless fetches from the main memory, generating useless memory traffic, and wasting bandwidth.

The third step chooses the local memory architecture template, in terms of the number of modules, and their type (spatial, temporal) according to the prevalent types of locality present in the application. The amount of locality of a certain type is determined both by the total size of the variables exhibiting that locality, and the intensity of the locality prediction. For instance if the spatial locality cluster contains large arrays, with high spatial density, it is beneficial to include a local memory module exploiting spatial locality.

The fourth step chooses the characteristics of the memory modules selected in the previous step. Since the total amount of local memory space is limited, we vary the relative sizes of the different memory modules, and the characteristics of each module such as line size, to trade-off hit ratio against memory bandwidth requirement.

The last step maps the variables to the local memory modules. The local memory modules cover mutually exclusive memory areas. Therefore, each variable will be serviced by only one local memory module, determined statically. We start from the initial mapping of the variables to local memory modules according to the clusters from Step 2. Since the original clustering in Step 2 is performed based on symbolic information only, without knowledge of the physical characteristics of the memory modules, we use profiling to fine-tune the mapping.

Due to the customized local memory architecture targeting separately the different types of locality in the application, the memory bandwidth is managed more judiciously, resulting in significant power savings, without sacrificing hit ratio. For more details on the PaLM algorithm and the customized cache architecture, please refer to [9].

## 6 Experiments

We present a set of experiments demonstrating the memory bandwidth and power reductions obtained by our Access Pattern based Local Memory Customization (PaLM) algorithm.

### 6.1 Experimental setup

In our experiments we use a processor architecture based on the SUN SPARC, with 8KB of local memory space. The applications have been compiled using gcc. The hit ratio and bandwidth have been computed using a version of our memory subsystem simulator based on shade [28] and SIMPRESS [16]. The power figures have been obtained with the WATTCH power simulator [2], assuming a technology of 0.35 micron.

The benchmarks are a combination of large applications, and smaller kernels from the multimedia, general purpose, and scientific domains. Vocoder is a GSM voice coding application (15K lines of code). Go, compress, and li are from the SPECINT95 benchmark suite (2K to 30K lines of code), while sor and madd are scientific kernels.

### 6.2 Results

Table 2 presents the traditional and customized local memory architectures, obtained using our Pattern Based Local Memory Customization for Low Power (PaLM) algorithm, along with the resulting hit ratios. In order to separate out the benefit of using specific local memory modules to target the different types of locality in the application, we compare our approach to the best traditional cache configuration for the given local memory size. The first column in Table 2 shows the benchmarks, the second column shows the best cache configuration for the traditional approach, and the third column shows the hit ratio for this architecture. The three numbers representing the cache architecture are the cache size in bytes, the cache line size in words (a word is 4 bytes), and the cache associativity. The fourth column shows the local memory configuration for out customized architecture, while the last column presents the corresponding hit ratio.

| bench | Traditional cache arch | | PaLM | |
|---|---|---|---|---|
| | organization | hit ratio | organization | hit ratio |
| go | 8K/8/2 | 95 | 2K/4/2, 6K/8/2 | 95 |
| compress | 8K/128/2 | 99 | 2K/8/2, 6K/128/2 | 99 |
| li | 8K/32/2 | 99.9 | 2K/4/2, 6K/128/2 | 99.9 |
| madd | 512/8/2 | 83 | 256/4/2, 256/8/2 | 94 |
| sor | 8K/64/2 | 99 | 2K/32/2, 6K/64/2 | 99 |
| vocoder | 512/8/2 | 99 | 256/4/2, 256/8/2 | 99 |

**Table 2. The traditional and customized local memory architectures and hit ratios.**

For most of the applications, the hit ratios in our approach are similar, or marginally better than in the traditional approach. For madd, the hit ratio is substantially improved, due to the fact that it contains a large number of stream accesses, and can benefit from the inclusion of the spatial cache to handle them, while keeping variables without spatial locality in the temporal cache.

Table 3 presents the bandwidth and power decrease generated by our approach. The second and third columns show the bandwidth and memory/interconnect power for the traditional cache architecture. The fourth column shows the bandwidth for our customized memory organization, while the fifth column presents the memory and interconnect power consumption for our customized architecture, including the cache control overhead due to the extra logic. The last column shows the percentile decrease in power generated by our customized local memory, compared to the traditional approach. The power reduction varies between -33% (representing a power overhead, for the li benchmark, where there where no opportunities for the customized memory to improve the locality match), and 77% (for madd, where the local memory customization could substantially reduce the memory bandwidth requirement). The power savings for the large benchmarks, while not as radical as for the kernels, are significant (18% to 31%). The average memory power reduction is 30%.

The large power savings are due to the customization of the local memory architecture, which separately targets the different locality types in the application, generating a better match between the application and the local memory. This effect is particularly significant in large applications containing variables with many different types of access patterns and locality, or after locality improving optimizations, which exacerbate the particular type of locality, and increase the need for a more focused local memory architecture.

| bench | traditional cache arch | | customized cache arch | | % power |
|---|---|---|---|---|---|
| | bandwidth | power | bandwidth | power | decrease |
| go | 0.32 | 0.832 | 0.26 | 0.67 | 23 |
| compress | 3.67 | 9.54 | 3.11 | 8.08 | 18 |
| li | 0.0156 | 0.04 | 0.0224 | 0.06 | -33 |
| madd | 2.5 | 6.5 | 1.41 | 3.66 | 77 |
| sor | 0.31 | 0.806 | 0.19 | 0.49 | 63 |
| vocoder | 0.024 | 0.062 | 0.018 | 0.047 | 31 |
| | | | | **average** | **30** |

**Table 3. The bandwidth and power reductions obtained by our Local Memory Customization Algorithm.**

## 7 Summary

We presented an approach for customization of the local memory architecture, to target the different types of locality in the application, and substantially reduce memory bandwidth and power consumption, without sacrificing performance.

Traditionally, the local memory architecture relied on a large cache to store all the variables in the application. However, especially in large real-life applications, different types of data exhibit different types of locality and access patterns, with different locality and bandwidth needs. The best the traditional architecture could do was to compromise the needs of

the different variables, by trading off temporal versus spatial properties of the cache. By using a temporal cache to store the variables with high temporal and low spatial locality, and a spatial cache to store the variables with high spatial locality, it is possible to finetune the local memory architecture to the locality needs of the access patterns in the application, and substantially reduce the main memory bandwidth and power consumption, without sacrificing performance.

We presented a set of experiments which show the bandwidth and power reduction obtained by our Local Memory Customization approach. The average power reduction was 30% (with similar or better cache hit ratios), over the best traditional configuration, for a set of large multimedia and general purpose applications, and scientific kernels.

Currently our work applies to temporal and spatial caches. Our on-going work evaluates this technique in the presence of other forms of caching, such as compiler-controlled on-chip memories, and prefetching.

## 8 Acknowledgments

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

## References

[1] L. Benini, A. Macii, E. Macii, M. Poncino, and R. Scarsi. Architectures and synthesis algorithms for power-efficient bus interfaces. *TCAD*, 19(9), 2000.

[2] D. Brooks, V. Tiwary, and M. Martonosi. Wattch: A framework for architectural-level power analysis and optimizations. In *ISCA*, 2000.

[3] D. Callahan, K. Kennedy, and A. Porterfield. Software prefetching. In *ASPLOS*, 1991.

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

[5] T-F. Chen and J-L. Baer. A performance study of software and hardware data prefetching schemes. In *ISCA*, 1994.

[6] K. Diefendorff and P. Dubey. How multimedia workloads will change processor design. In *Micro*, 1997.

[7] S. Rixner et. al. Memory access scheduling. In *ISCA*, 2000.

[8] A. Gonzales, C. Aliagas, and M. Valero. A data cache with multiple caching strategies tuned to different types of locality. In *International Conference on Supercomputing (ICS)*, 1995.

[9] P. Grun, N. Dutt, and A. Nicolau. Early local memory exploration for low power. Technical report, University of California, Irvine, 2000.

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

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

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

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

[14] A. Huang and J. Shen. A limit study of local memory requirements using value reuse profile. In *Annual Symposium of Microarchitecture*, 1995.

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

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

[17] C. Kulkarni, F. Catthoor, and H. de Man. Code transformations for low power caching in embedded multimedia processors. In *Intnl. Parallel Processing Symposium (IPPS), Orlando, FL*, 1998.

[18] D. Lee, P. Crowley, J-L. Baer, T. Anderson, and B. Bershad. Execution characteristics of desktop applications on windows nt. In *ISCA*, 1998.

[19] C. Luk and T. Mowry. Memory forwarding: Enabling aggressive layout optimizations by guaranteeing the safety of data relocation. In *ISCA*, 1999.

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

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

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

[23] P. R. Panda, N. D. Dutt, and A. Nicolau. Low power memory mapping through reducing address bus activity. In *IEEE Transactions on VLSI*.

[24] Betty Prince. *High Performance Memories, New Architecture DRAMs and SRAMs evolution and function*. Wiley, West Sussex, 1996.

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

[26] P. Ranganathan, S. Adve, and N. Jouppi. Performance of image and video processing with general-purpose processors and media isa extensions. In *ISCA*, 1999.

[27] P. Ranganathan, S. Adve, and N. Jouppi. Reconfigurable caches and their application to media processing. In *ISCA*, 2000.

[28] Sun Microsystems, The Shade Simulator. *sw.sun.com/shade*.

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

[30] M. Wolf and M. Lam. A data locality optimizing algorithm. In *PLDI*, 1991.

[31] 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.