Application-Specific Memory Management for Embedded Systems Using Software-Controlled Caches

Derek Chiou, Prabhat Jain, Larry Rudolph, and Srinivas Devadas
Department of EECS
Massachusetts Institute of Technology
Cambridge, MA 02139

ABSTRACT

We propose a way to improve the performance of embedded processors running data-intensive applications by allowing software to allocate on-chip memory on an application-specific basis. On-chip memory in the form of cache can be made to act like scratchpad memory via a novel hardware mechanism, which we call *column caching*. Column caching enables dynamic cache partitioning in software, by mapping data regions to a specified sets of cache "columns" or "ways." When a region of memory is exclusively mapped to an equivalent sized partition of cache, column caching provides the same functionality and predictability as a dedicated scratchpad memory for time-critical parts of a real-time application. The ratio between scratchpad size and cache size can be easily and quickly varied for each application, or each task within an application. Thus, software has much finer software control of on-chip memory, providing the ability to dynamically tradeoff performance for on-chip memory.

1. INTRODUCTION

As time-to-market requirements of electronic systems demand ever faster design cycles, an ever increasing number of systems are built around a programmable embedded processor that implements an ever increasing amount of functionality in firmware running on the embedded processor. The advantages of doing so are twofold: software is simpler to implement than a dedicated hardware solution and software can be easily changed to address design errors, late design changes and product evolution[13] [12]. Only the most time-critical tasks need to be implemented in hardware.

On-chip memory, in the form of cache, scratchpad SRAM, (and more recently) embedded DRAM or some combination of the three, is ubiquitous in programmable embedded systems to support software and to provide an interface between hardware and software. Most systems have both cache and scratchpad memory on-chip since each addresses a different need. Caches are transparent to software since they are accessed through the same address space as the larger backing storage. They often improve overall software performance but are unpredictable. Although the cache replacement hardware is known, predicting its performance depends on accurately predicting past and future reference patterns. Scratchpad memory is addressed via an independent address space and thus must be managed explicitly by software, oftentimes a complex and cumbersome problem, but provides absolutely predictable performance. Thus, even though a pure cache system may perform better overall, scratchpad memories are necessary to guarantee that critical performance metrics are always met.

Of course, both caches and scratchpad memories should be available to embedded systems so that the appropriate memory structure can be used in each instance. A static division, however, is guaranteed to be suboptimal as different applications have different requirements. Previous research has shown that even within a single application, dynamically varying the partitioning between cache and scratchpad memory can significantly improve performance[11].

We propose a way to dynamically allocate cache and scratchpad memories from a common pool of memory. In particular, we propose column caching[2], a simple modification that enables software to dynamically partition a cache into several distinct caches and scratchpad memories at a *column* granularity. In our reference implementation, each "way" of an n-way set-associative cache is a *column*. By exclusively allocating a region of address space to an equal-sized region of cache, column caching can emulate scratchpad memory. Column caching only restricts data placement within the cache during replacement, all other operations are unmodified.

Careful mapping can potentially reduce or eliminate replacement errors, resulting in improved performance. It not only enables a cache to emulate scratchpad memory, but separate spatial/temporal caches, a separate prefetch buffer, separate write buffers and other traditional, statically-partitioned structures within the general cache as well. The rest of this paper describes column caching and how it might be used.

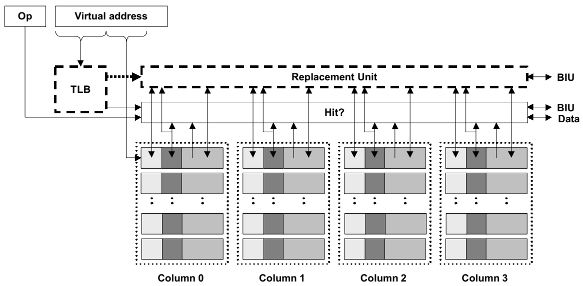

Figure 1: Basic Column Caching. Three modifications to a set-associative cache, denoted by dotted lines in the figure, are necessary: (i) augmented TLB to hold mapping information, (ii) modified replacement unit that uses mapping information and (iii) a path between the TLB and the replacement unit that carries that information.

2. COLUMN CACHING

The simplest implementation of column caching is derived from a set-assocative cache where lower-order bits are used to select a *set* of cache-lines which are then associatively searched for the desired data. If the data is not found (a cache miss), the replacement algorithm selects a cache-line from the selected set.

During lookup, a column cache behaves exactly as a standard set-assocative cache and thus incurs no performance penalty on a cache hit. Rather than allowing the replacement algorithm to always select from any cache-line in the set, however, column caching provides the ability to restrict the replacement algorithm to certain *columns*. Each column is one "way", or bank, of the n-way set-associative cache (Figure 1). Embedded processors such as the ARM[5] are highly set-associative to reduce power consumption, providing a large number of columns. A bit vector specifying the permissible set of columns is generated and passed to the replacement unit.

A modification to the bit vector *repartitions* the cache. Since every cache-line in the set is searched during every access, repartitioning is graceful and fast; if data is moved from one column to another (but always in the same set), the associative search will still find the data in the new location. A memory location can be cached in one column during one cycle, then re-mapped to another column on the next cycle. The cached data will not move to the new column instantaneously, but will remain in the old column until it is replaced. Once removed from the cache, it will be cached in a column to which it is mapped the next time it is accessed.

Column caching is implemented via three small modifications to a set-associative cache (Figure 1). The TLB must be modified to store the mapping information. The replacement unit must be modified to respect TLB-generated restrictions of replacement cache-line selection. A path to carry the mapping information from the TLB to the replacement unit must be provided. Similar control over the cache already exists for uncached data, since the cached/uncached bit resides in the TLB.

2.1 Partitioning and Repartitioning

Implementation is greatly simplified if the minimum mapping granularity is a page, since existing virtual memory translation mechanisms including the ubiquitous translation-look-aside-buffers (TLB) can be used to store mapping information that will be passed to the replacement unit. TLB's, accessed every memory reference, are designed to be fast in order to minimize physical cache access time. Partitioning is supported by simply adding column caching mapping entries to the TLB data structures and providing a data path from those entries to the modified replacement unit. Therefore, in order to remap pages to columns, access to the page table entries is required.

Mapping a page to a cache partition represented by a bit vector is a two phase process. Pages are mapped to a *tint* rather than to a bit vector directly. A tint is a virtual grouping of address spaces. For example, an entire streaming data structure could be mapped to a single tint, or all streaming data structures could be mapped to a single tint, or just the first page of several data structures could be mapped to a single tint. Tints are independently mapped to a set of columns, represented by a bit vector; such mappings can be changed quickly. Thus, tints, rather than bit vectors, are stored in page table entries.

The tint level-of-indirection is introduced (i) to isolate the user from machine-specific information such as the number of columns or the number of levels of the memory hierarchy and (ii) to make re-mapping easier.

2.2 Using Columns As Scratchpad Memory

Column caching can emulate scratchpad memory within the cache by dedicating a region of cache equal in size to a region of memory. No other memory regions are mapped to the same region of cache. Since there is a one-to-one mapping, once the data is brought into the cache it will remain there. In order to guarantee performance, software can perform a load on all cache-lines of data when remapping as is required with a dedicated SRAM. That memory region then behaves like a scratchpad memory. If and when the scratchpad memory is remapped to a different use, the data is automatically copied back (if backing RAM is provided).

2.3 Impact of Column Caching on Clock Cycle

The modifications required for column caching are limited to the cache replacement unit which is not on the critical path. In realistic systems, data requested from L1 cache to main memory takes at least three cycles, but generally more, to return. The exact replacement cache-line does not need to be decided until the data returns, giving the replacement algorithm at least three cycles to make a decision, which should easily be sufficient for the minor additions to the replacement path.

3. PREDICTABILITY IN MULTITASKING

Column caching enables predictable performance within a multitasking environment where multiple jobs are executing. Consider three compression (gzip) jobs simultaneously executing on one processor and each having access to the cache. To understand what is happening, we only present the performance measurement of a one gzip process (referred to as job A in the rest of the discussion) in this mixture. We present the results in terms of clocks per instruction (CPI) which is inversely correlated with performance – a lower CPI means higher performance. To compute the CPI, we assume a 20 cycle latency to memory and that 25% of instructions are memory operations. Figure 2, shows the variation in job A's CPI when the time quantum is varied. Results for both a standard cache and a mapped column cache are presented. The two sets of plots correspond to different sized (16K and 128K) caches.

Each point in this plot was generated by choosing a time quantum, and performing a round-robin schedule of the three gzip jobs, A, B and C. There are two cases: (i) each job gets to use the entire cache while it is running (standard cache), and (ii) each job uses only its assigned columns (column cache). For the column cache, the critical job is permitted to use the entire cache, while the other two jobs are restricted to using only a quarter of the cache. In the standard cache case, there is a significant difference in the CPI for job A, as the time quantum varies. This variation is caused mainly by the cache hit rate for job A being affected by intervening cache accesses due to jobs B and C. The number of such accesses is dependent on the time quantum. Once column caching is introduced and most of job A's data is protected from replacement by jobs B and C's data, then the CPI of job A is significantly less sensitive to the time quantum. Job A was considered critical, and it was exclusively assigned a large fraction of the cache, hence the hit rate for job A is higher. Therefore, the CPI is significantly smaller for small time quanta. Of course when the time quantum is really large, we effectively have batch processing and the CPI's are virtually the same for the top two plots. Overall system throughput may actually decrease due to an over allocation of resources to a set of critical jobs, but the performance of those critical jobs is generally higher and has much less variation.

One may argue that the time quantum could be fixed for predictability, but in reality due to interrupts and exceptions the effective time quantum can vary significantly during the time that a job is running simultaneously with other jobs. Thus, column caching can improve performance of a critical job as well as significantly reduce performance variation even in the presence of interrupts or varying time quanta.

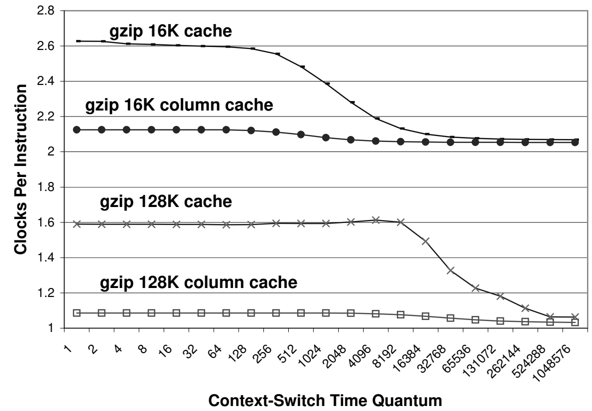

Figure 2: Column caching provides predictable and superior performance to a standard cache over a wide range of scheduling time quanta. The clocks per instruction is a measure of performance, the smaller the number the better. Except for very long time quantum periods, column caching provides superior performance. Also note that the performance of column caching is much less sensitive to time quantum times, as seen from the nearly horizontal curves for column caching.

4. RELATED WORK

4.1 Cache Mechanisms

The idea of statically-partitioned caches is not new. The most common example are separate instruction and data caches. Some existing and proposed architectures support a pair of caches, one for spatial locality and one for temporal locality [14, 15, 5, 1, 7, 4]. These designs statically divide the two caches. Other processors support locking of data into the cache[3, 9], but do not include a way to tell if the desired data is in the cache. Sun Microsystems Corporation patented a mechanism [10] very similar to column caching that allows partitioning of a cache between processes at cache column granularity by providing a bit mask associated with the running process, limiting it to partitioning the cache between processes.

A subset of column caching abilities is achievable without hardware support by page coloring, achieved by intelligently mapping virtual pages to physical pages. Column caching, however, is much faster at repartitioning (page coloring requires a memory copy), uses set-associative caches better and enabling such abilities as mapping a large contiguous region of address space to a small region in the cache (useful for memory-mapped devices).

4.2 Memory Exploration in Embedded Systems

Cache memory issues have been studied in the context of embedded systems. McFarling presents techniques of code placement in main memory to maximize instruction cache hit ratio [8, 16]. A model for partitioning an instruction cache among multiple processes has been presented [6].

Panda, Dutt and Nicolau present techniques for partitioning on-chip memory into scratchpad memory and cache [11]. The presented algorithm assumes a fixed amount of scratchpad memory and a fixed-size cache, identifies critical variables and assigns them to scratchpad memory. The algorithm can be run repeatedly to find the optimum performance point.

5. CONCLUSIONS

The work described in this paper represents a confluence of two observations. The first observation is that given heterogeneous applications with data streams that have significant variation in their locality properties, it is worthwhile to provide finer software control of the cache so the cache can be used more efficiently. To this end, we have proposed a column caching mechanism that enables cache partitioning so data with different locality characteristics can be isolated for improved performance. The second observation is that columns can emulate scratchpad memory which is used extensively to improve predictability in embedded systems. A significant benefit of column caching is that through the execution of a program, the data stored in columns can be explicitly managed as in a scratchpad or can be implicitly managed as in a cache and that the management can change dynamically and at very small intervals.

Acknowledgements: This paper describes research done at the Laboratory for Computer Science of the Massachusetts Institute of Technology. Funding for this work is provided in part by the Advanced Research Projects Agency of the Department of Defense under the Air Force Research Laboratory contract F30602-99-2-0511.

6. REFERENCES

[1] K. Asanovic. *Vector Microprocessors*. PhD thesis, University of California at Berkeley, May 1998.

[2] D. T. Chiou. *Extending the Reach of Microprocessors: Column and Curious Caching*. PhD thesis, Department of EECS, MIT, Cambridge, MA, Sept. 1999.

[3] Cyrix. *Cyrix 6X86MX Processor*, May 1998.

[4] G. Faanes. A CMOS Vector Processor with a Custom Streaming Cache. In *Hot Chips 10*, August 1998.

[5] Intel. *Intel StrongARM SA-1100 Microprocessor*, April 1999.

[6] Y. Li and W. Wolf. A Task-Level Hierarchical Memory Model for System Synthesis of Multiprocessors. In *Proceedings of the* 34^{th} *Design Automation Conference*, pages 153–156, June 1997.

[7] B. Lynch and G. Lauterbach. UltraSPARC III: A 600 MHz 64-bit Superscalar Processor for 1000-Way Scalable Systems. In *Hot Chips 10*, 1998.

[8] S. McFarling. Program Optimization for Instruction Caches. In *Proceedings of the* 3^{rd} *Int'l Conference on Architectural Support for Programming Languages and Operating Systems*, pages 183–191, April 1989.

[9] Motorola. *MPC8240 Integrated Processor User's Manual*, July 1999.

[10] B. Nayfeh and Y. A. Khalidi. Us patent 5584014: Apparatus and method to preserve data in a set associative memory device, Dec. 1996.

[11] P. R. Panda, N. Dutt, and A. Nicolau. *Memory Issues in Embedded Systems-on-Chip: Optimizations and Exploration*. Kluwer Academic Publishers, 1999.

[12] P. G. Paulin, C. Liem, T. C. May, and S. Sutarwala. DSP Design Tool Requirements for Embedded Systems: A Telecommunications Industrial Perspective. *Journal of VLSI Signal Processing*, 9(1/2):23–47, January 1995.

[13] J. V. Praet, G. Goossens, D. Lanneer, and H. D. Man. Instruction Set Definition and Instruction Selection for ASIPs. In *Proceedings of the 7th IEEE/ACM International Symposium on High-Level Synthesis*, May 1994.

[14] F. Sánchez, A. González, and M. Valero. Software Management of Selective and Dual Data Caches. In *IEEE Computer Society Technical Committee on Computer Architecture: Special Issue on Distributed Shared Memory and Related Issues*, pages 3–10, Mar. 1997.

[15] M. Tomasko, S. Hadjiyiannis, and W. Najjar. Experimental Evaluation of Array Caches. In *IEEE Computer Society Technical Committee on Computer Architecture: Special Issue on Distributed Shared Memory and Related Issues*, pages 11–16, Mar. 1997.

[16] H. Tomiyama and H. Yasuura. Code Placement Techniques for Cache Miss Rate Reduction. *ACM Transactions on Design Automation of Electronic Systems*, 2(4):410–429, October 1997.