Reconfigurable Caches and their Application to Media Processing *

Parthasarathy Ranganathan
Department of Electrical and Computer Engineering
Rice University
Houston, Texas
parthas@rice.edu

Sarita Adve
Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, Illinois
sadve@cs.uiuc.edu

Norman P. Jouppi
Western Research Laboratory
Compaq Computer Corporation
Palo Alto, California
jouppi@pa.dec.com

ABSTRACT

High performance general-purpose processors are increasingly being used for a variety of application domains – scientific, engineering, databases, and more recently, media processing. It is therefore important to ensure that architectural features that use a significant fraction of the on-chip transistors are applicable across these different domains. For example, current processor designs often devote the largest fraction of on-chip transistors (up to 80%) to caches. Many workloads, however, do not make effective use of large caches; e.g., media processing workloads which often have streaming data access patterns and large working sets.

This paper proposes a new reconfigurable cache *design. This design enables the cache SRAM arrays to be dynamically divided into multiple partitions that can be used for different processor activities. These activities can benefit applications that would otherwise not use the storage allocated to large conventional caches. Our design involves relatively few modifications to conventional cache design, and analysis using a modification of the CACTI analytical model shows a small impact on cache access time. We evaluate one representative use of reconfigurable caches – instruction reuse for media processing. We find this use gives IPC improvements ranging from 1.04X to 1.20X in simulation across eight media processing benchmarks.*

*This work is supported in part by an IBM Partnership award, Intel Corporation, the National Science Foundation under Grant No. CCR-9502500, CDA-9502791, CDA-9617383, and CCR-0096126 and the Texas Advanced Technology Program under Grant No. 003604-025. Sarita Adve is also supported by an Alfred P. Sloan Research Fellowship.

1. INTRODUCTION

Current high performance general-purpose processors are used for a variety of application domains, including scientific, engineering, transaction processing, and decision support. More recently, media processing applications have received significant attention. Media processing applications have challenging computational requirements and could use orders-of-magnitude higher performance than available today. In the past, they were run on specialized DSP processors or ASICs. However, the benefits of general-purpose processors in terms of easier programmability, upgradability, and higher performance growth curves argue for the increased use of such systems for media processing.

Several quantitative characterizations have shown that applications from different domains exhibit different characteristics [5, 11, 15, 19]. A handicap for general-purpose systems is that they must perform well enough across all such characteristics. As these systems are used for an increasingly wide variety of applications, a "one-size-fits-all" design philosophy will be inadequate. Current system designs often include special features targeted to certain key applications; however, often, these features are implemented in a rigid manner and their resources are wasted for applications that cannot directly utilize them. For example, the use of large caches is a common trend across general-purpose systems, sometimes consuming up to 80% of the total transistor budget and up to 50% of the die area [9]. While large caches are effective for a variety of conventional workloads, they are often ineffective for media processing applications because of the streaming nature of data accesses and the large working sets in these applications [19].

An alternative design philosophy is to build some flexibility in the system so that features that use a large number of resources can be used in different ways by different applications. We apply this philosophy to the design of caches and propose a new cache organization that we call *reconfigurable caches*. The idea of reconfigurable architectures itself is not new; however, conventional reconfigurable architectures typically involve large changes in both hardware and software. In contrast, our reconfigurable cache design continues to use a conventional cache design with minor hardware and software changes.

We propose a reconfigurable cache organization that allows the on-chip SRAM to be dynamically divided into different partitions that can be assigned to different processor activities other than conventional caching. For example, the partitions could be used as hardware look-up tables for techniques such as instruction reuse and hardware prefetching, or as storage area for prefetched information, or as compiler-controlled memory. Thus, the cache SRAM storage can benefit applications that would not otherwise exploit large conventional caches.

In our model the reconfigurability is implemented in custom hardware at design time. This is in contrast to an approach that might use hardware similar to field-programmable gate-arrays (FPGAs). While useful in many applications, the ability to dramatically change the programming of FPGAs in the field results in FPGA clock cycle times that are typically 4 to 6 times slower than that obtained in full-custom microprocessors. Instead, we support a limited number of possible configurations (e.g., two or three) which are incorporated into the circuit design and verified as part of the original microprocessor design.

The paper first discusses several applications of reconfigurable caches (Section 2). It then discusses the organization (Section 3) and design (Section 4) of the reconfigurable cache. Two key design challenges are determining how to partition the cache SRAM array and how to address the different partitions as they change, without significantly affecting access time. The primary design proposed in this paper addresses these challenges by exploiting the design and implementation of conventional set associative caches using two insights. First, different *ways* of a set associative cache can be used to form different partitions, enabling the use of the conventional cache addressing mechanism with minor modifications. Second, conventional cache SRAM arrays are also implemented as multiple subarrays to reduce and balance the length of wordlines and bitlines. We continue to use such a subarray organization for our partitions, possibly with different number and dimensions of the subarrays. We extend the CACTI analytical model [25] to show that our modifications do not significantly impact the cache access time for several configurations.

To provide quantitative evidence of the benefits of reconfigurable caches, we evaluate instruction reuse for media processing as a representative application of a reconfigurable cache (Section 5). We use detailed simulation to study eight media processing benchmarks. Our results show that using reconfigurable caches for instruction reuse achieves improvements in IPC ranging from 1.04X to 1.20X across our benchmarks.

2. POTENTIAL APPLICATIONS OF RECONFIGURABLE CACHES

Some of the possible applications for reconfigurable caches are discussed below. We specifically discuss how these applications are relevant to the domain of media processing; however, codes from other domains can benefit from these as well.

Hardware optimizations using lookup tables or buffers. Several hardware optimizations have been proposed that require maintaining lookup tables or buffers, where the effectiveness of the optimization improves significantly with larger table sizes. For example, value prediction, memoization, and instruction reuse have recently been studied to exploit redundancy in computation in the SPEC benchmarks [7, 12, 13, 22, 23, 24]. Other optimizations that require large lookup tables or buffers include coherence prediction, memory disambiguation prediction, compression-based branch prediction, hardware prefetching (where lookup tables are used to store information for address prediction), and dynamic optimizations triggered by performance information collected and stored in tables at runtime. Several of these techniques have been reported to have the capacity to perform better with larger lookup table spaces [22, 23]. The lookup tables and buffers for these optimizations could be implemented in a partition of a reconfigurable cache instead of using other valuable chip area. Section 5 studies one such technique, instruction reuse, with reconfigurable caches to address the computation bottleneck in media processing workloads.

Software and hardware prefetched data. Software and hardware prefetching are widely used techniques to hide memory latency. However, if the prefetched data is fetched too far in advance, it can pollute the cache replacing other useful data or be replaced before use by a demand access, eliminating any performance benefits. On the other hand, prefetches that occur too late do not fully hide the latency. Therefore, prefetching techniques need to strike a careful balance when scheduling the prefetches, but are often unsuccessful in doing so. With reconfigurable caches, a separate partition can be used to prefetch data early while avoiding the problem of cache pollution or replacement of prefetched data. Such an application of reconfigurable caches could be particularly useful with media processing benchmarks which often have streaming behavior [19].

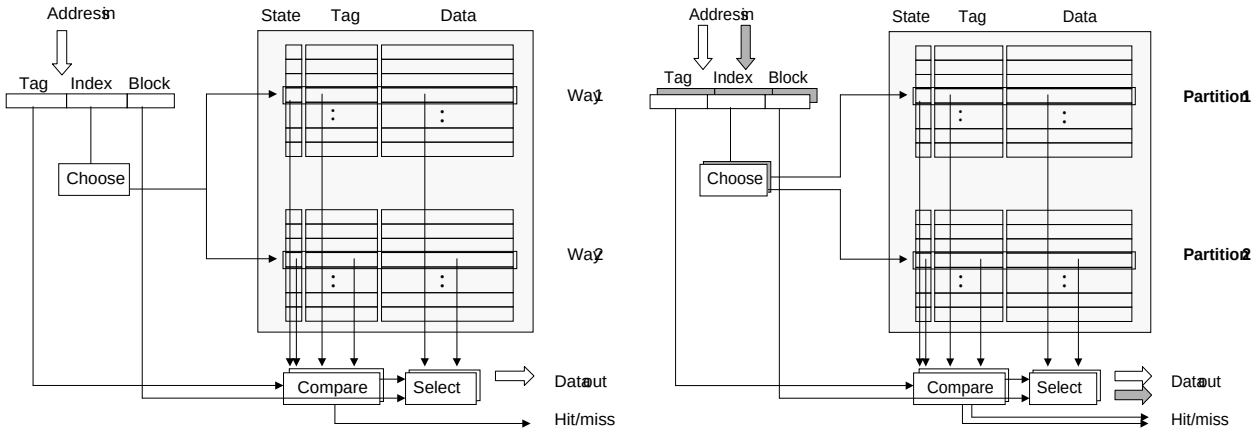
Compiler or application controlled memory. A partition of a reconfigurable cache could be configured as compiler or application controlled memory. As discussed in [4], the compiler could use such memory as a scratch area for spill code to achieve performance benefits. Alternatively, this area can be used by system code or device drivers as a separately addressable buffer area. Such a use may also be beneficial in ensuring real-time requirements of media applications with general-purpose processors. Many DSP processors hardwire their on-chip SRAM to be used as memory (as opposed to caches) to ensure predictability of memory latencies [6]. Cache line locking (as in the Cyrix MediaGX processor [14]) or controlled cache line replacement (as with malleable caches [3]) can provide the same functionality.

3. CACHE ORGANIZATION

Reconfigurable caches require a cache organization that allows the on-chip SRAM cache storage to be dynamically partitioned and reused for other processor activities requiring storage. There are several aspects to such an organization with multiple design options and tradeoffs, discussed next.

3.1 Partitioning and Addressing

The key challenge in designing a reconfigurable cache is to devise a mechanism to divide the SRAM storage into different (possibly variable-sized) partitions, and to efficiently



(a) Conventional cache organization

(b) Reconfigurable cache organization

Figure 1: Associativity-based partitioning organization for reconfigurable caches

be able to address these partitions. In particular, the addressing scheme must efficiently adapt to dynamic resizing of the partition sizes. Below, we first discuss a scheme based on cache associativity that is used in the rest of this paper. We then discuss an alternative that does not rely on cache associativity.

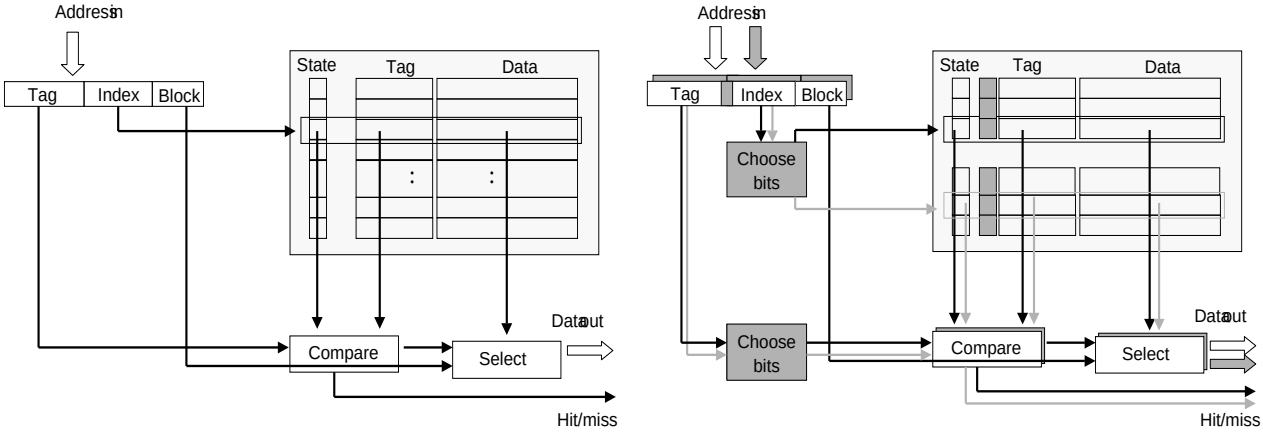
Associativity-based partitioning. Our primary design exploits set-associativity in current cache organizations. To understand the design, we briefly revisit a conventional set-associative cache organization. Figure 1(a) depicts the block diagram for the (conceptual) organization of a single-ported 2-way set-associative conventional cache. An N-way set associative cache is divided into N data and tag arrays, and each of these N pairs is referred to as a *way*. The index part of the input address is used to index all the ways of the data and tag array. The tag part of the input address is sent to the comparators of all the ways to determine if there is a match with any of the tags read from the tag array. If there is a match, a hit is signaled on the valid output line and data from the corresponding way of the data array is sent onto the output data lines.

The reconfigurable cache builds on the above organization in a natural way, as depicted in Figure 1(b). We divide the reconfigurable cache into partitions at the granularity of the *ways* of the conventional cache, exploiting the conceptual division into ways already present in a conventional cache. For example, at the finest granularity, a 4-way set-associative 1MB cache can be dynamically reconfigured into 4 partitions of 256KB each, with each way in a separate partition. At coarser granularities, a partition of such a cache may contain two ways (for a total of 512KB organized as a two-way set-associative partition) or three ways (for a total of 768KB organized as a three-way set associative partition). In each case, the same bits of the address field would be used as tag, index, and block offset bits. In the 1MB cache example above, assuming 64 byte cache lines, the last 6 bits of the address are used for the block offset, the next 12 to index each way, and the remaining for the tag. The only changes to the conventional cache organization are as follows:

- *Multiple input and output paths.* A reconfigurable cache with up to N partitions must accept N input addresses and generate N output data elements with N hit/miss signals (one for each partition). Section 4 discusses a design that can achieve this without increasing the number of cache ports.
- *Cache status register.* The current partitioning of the cache controls which of the N input addresses is used to index a specific way and to perform the tag match at the comparator of that way. Similarly, for the ways that produce data hits, the current partitioning determines which of the N output data paths should get this data and which of the N hit/miss lines should be signaled as a hit. A special hardware register called the *cache status register* is therefore maintained to track the number and sizes of the partitions and control the routing of the above signals. The cache status register is part of the processor state and needs to be preserved across context switches similar to any other control register.

From the processor's viewpoint, the various partitions can be addressed either implicitly for internal hardware activities (e.g., value prediction lookup tables) or explicitly for software-controlled use (e.g., compiler-controlled memory). The latter would require some ISA support to indicate the correct partition that the memory accesses need to be routed to (e.g., an approach similar to the address space identifiers could be used).

The above set-associativity based partitioning approach has at least three advantages. First, it requires only small changes to the current well-understood set-associative cache organization. Second, the mechanism for addressing the cache arrays scales well with the dynamic repartitioning of the cache. Third, this organization keeps requests for the different partitions isolated from each other, and so does not introduce any additional contention for the SRAM ways. However, the key drawback with this approach is that the number and granularity of the partitions are limited by the associativity of the cache. Larger or smaller granularity partitioning needs a higher-order associativity of the base cache which could increase cache access time and tag storage space. An



(a) Conventional cache organization (b) Reconfigurable cache organization

Figure 2: Overlapped wide-tag partitioning organization for reconfigurable caches

alternative organization that is slightly more complex but does not suffer from the above drawback is discussed next.

Overlapped wide-tag partitioning. The overlapped wide-tag partitioning approach does not require a set-associative cache organization. The dark-shaded regions in Figure 2(b) indicate the additional changes to be made to the conventional direct-mapped cache organization in Figure 2(a). Since the number of bits in the index and tag fields of the address vary based on the size of the partition, the size of the tag array in the cache SRAM also needs to change dynamically with the size of the partitions. The wide-tag organization extends the current SRAM tag array to account for the maximum variation in the tag size with different partition sizes. For example, for a direct-mapped 1MB cache which supports multiple partitions each of which is at least 256KB large, the tag array would be extended by an additional two bits to support the the maximum possible tag length required by the 256KB partitions. Though the cache partitions can potentially be any size, we limit them to be powers of two to enable simpler decoding. When the data is read from the cache, the additional logic shown in Figure 2(b) ensures that only the bits of the index and tag corresponding to the size of the particular partition are used for the cache access. This logic can be relatively simple.

3.2 Maintaining Data Consistency

Reconfigurable caches need a mechanism to ensure that after reconfiguration, the data belonging to a particular processor activity resides only in the partition associated with that particular activity. We discuss two approaches to ensure this below.

Cache scrubbing. This approach uses the partitioning information in the cache status register to ensure that at the time of reconfiguration, appropriate data gets moved between partitions or the data is moved from the cache to lower levels of memory. Such a *cache-scrubbing* approach requires examining all the locations of the cache to check for their validity and performing suitable actions on valid data at reconfiguration. For example, for a partition transformed from a cache to another activity, valid data needs to be sent to another partition used as cache or to lower levels of memory. Some applications such as compiler-controlled memory may require intervention at the operating system level to ensure that data is kept consistent. For partitions used as lookup tables for speculation, data consistency is not as critical an issue; not initializing the data to the correct values would just result in a longer "cold-start" period. Cache scrubbing can potentially incur a high reconfiguration overhead, but can be acceptable in cases of infrequent reconfiguration such as at the start of the application. For example, for the 128KB L1 cache with 64 byte cache lines in Section 5, assuming a fully pipelined 20-cycle L2 cache and support for 12 outstanding writebacks, the time taken to write all the 2048 cache lines to the L2 cache is less than 3500 cycles. In practice only a smaller fraction of these 2048 cache lines will have to be written back to the cache.

Lazy transitioning. In some cases, more frequent reconfiguration may be desirable (due to frequent context switches between applications or aggressive adaptive reconfiguration for the same application). For such cases, an alternate scheme is possible where data is lazily moved into its correct partition only when it is accessed. For such a scheme, the state information associated with a cache line needs to be augmented with information on the processor activity associated with that line. For example, for 4 partitions for the 128KB L1 cache in Section 5, if different activities are associated with each partition, this would require about 512 bytes of extra state storage. In addition to the normal tag lookup, this technique requires the processor activity state to also be validated before the data is considered a hit. On a miss in the appropriate partition, other partitions need to be checked to determine if the data is still in those partitions. This approach can reduce the high overhead associated with moving large amounts of data at reconfiguration time and increase the data hit rates in the partitions; however, these benefits come at the expense of increased state storage, a more complex implementation, and possible increased contention for the SRAM partitions.

3.3 Reconfiguration Policy and Detection

The third issue that needs to be addressed is the policy and detection mechanism for when to reconfigure. Repartitioning may occur infrequently (e.g., just once at the start of

Design Issue	Options	Used in this paper
Partitioning mechanism	Overlapped wide-tag vs. Associativity-based	Associativity-based
Address generation for non-cache partitions	Hardware vs. Software generated	Hardware generated
Data consistency	Cache scrubbing vs. Lazy transitioning	Cache scrubbing
Repartitioning policy	Frequent vs. Infrequent	Infrequent
Detection mechanism for reconfiguration	Software vs. Hardware control	Software control
Reconfigurable cache level	L1, L2, or lower levels	L1

Table 1: Reconfigurable cache organization choices.

the application) or frequently (e.g., at the beginning of certain loops), depending on the characteristics of the application and the processor activities for which the reconfigurable cache is used. The mechanism to detect when to reconfigure can be software or hardware controlled. A software-controlled approach can expose the cache status register to the code-generator (user or compiler) which can use information about the program behavior to invoke appropriate reconfiguration at the appropriate points in the program. Alternately, a hardware-controlled approach could use hardware performance monitoring support (e.g., DCPI [2]) to automatically decide when and how to change the partitions.

3.4 Reconfigurable Cache Level

The final issue with the reconfigurable cache organization is the level that is reconfigurable in a multi-level cache hierarchy. The cache organization described above does not preclude its application to any level. Tradeoffs in terms of the size, granularity, access time, and usage of the partitions will determine the level to partition.

3.5 Options Used in This Paper

Table 1 summarizes the various aspects of the reconfigurable cache organization and gives the configuration we study in this paper. Specifically, we study a hardware-addressed associativity-based partitioning approach with software control for infrequent reconfiguration and cache scrubbing to ensure consistency of data. We apply this configuration to the L1 cache in Section 5. This configuration represents a simple organization that is likely to achieve most of the performance benefits for the applications discussed in Section 2.

4. DESIGN AND IMPLEMENTATION

Sections 4.1–4.3 first discuss a more detailed implementation of the reconfigurable cache organization discussed in Section 3.5. Section 4.4 then uses a modification of the CACTI model for a detailed timing analysis of a reconfigurable cache.

4.1 Conventional Cache Implementations

Figure 3 shows a typical implementation of the internal structure of a conventional SRAM cache. The numbered shaded blocks refer to the changes needed for reconfigurable caches and are discussed in subsequent subsections.

The key components are the data and tag arrays. The arrays consist of the storage cells, the horizontal wordlines that enable a single row of cells, and the vertical bitlines that transfer data from the selected cell of a column. A straightforward implementation of the data (or tag) array would have S rows and $8BA$ columns of storage cells, where S is the number of sets in the cache, B is the line size in bytes, and A is the associativity. This implementation, however, would incur large cache access times because of the long and unbalanced wordline and bitline delays. Instead, existing implementations divide both the data and tag arrays into multiple subarrays, as shown in Figure 3. In this paper, we use the CACTI timing model [20, 25] to identify the optimal values for the dimensions of the subarrays. Note that a row of a subarray can have cells from multiple ways of the same set.

Other components of the cache and the various delay components that comprise the operation of a cache (as modeled by CACTI) are as follows. A decoder for each SRAM subarray first decodes the address (incurring a *decoder delay*) and selects the appropriate subarray row by driving one wordline in the array (*wordline delay*); only one wordline in each subarray can be high at a time. Each memory cell along the selected row is associated with a pair of bitlines that is initially precharged high. When a wordline goes high, the memory cell in that row pulls down one of its bitlines which determines the value stored in the cell (*bitline delay*). If a bitline is shared between multiple columns or subarrays, a column multiplexor is used to choose the relevant bit lines for the particular SRAM access (*column multiplexor delay*). A sense amplifier is used to detect which line goes low and determine the value in the memory cell (*sense amplifier delay*). For the tag array, the information read from the various ways is compared to the tag bits of the address (*comparator delay*). The results of these comparisons are used to drive a valid output signal that indicates whether there is a hit or a miss (*valid output driver delay*). If the cache access was a hit, in a set associative cache, the results of the comparisons also choose the correct data from the data array (*select* and *data output delays*). Depending on the cache configuration, the critical path could be either through the tag array or through the data array. The cache access time is the sum of the delays on the critical path (obtained by adding the appropriate delay components discussed above).

4.2 SRAM Partitioning for Reconfigurability

As discussed above, current cache implementations already partition the cache array into multiple subarrays. The partitioning required for reconfigurable caches can therefore naturally exploit this structure. A key difference between a conventional and reconfigurable cache, however, is that different partitions in the latter need to be indexed by different addresses. Therefore, the cache ways corresponding to different partitions must be implemented in physically different subarrays of the cache; i.e., there must be at least as many subarrays as the maximum number of partitions. In contrast, in a conventional set-associative cache, for a given set, all or some of the ways of the set can potentially be implemented within a single subarray. Thus, it may not be possible to implement a reconfigurable cache with the

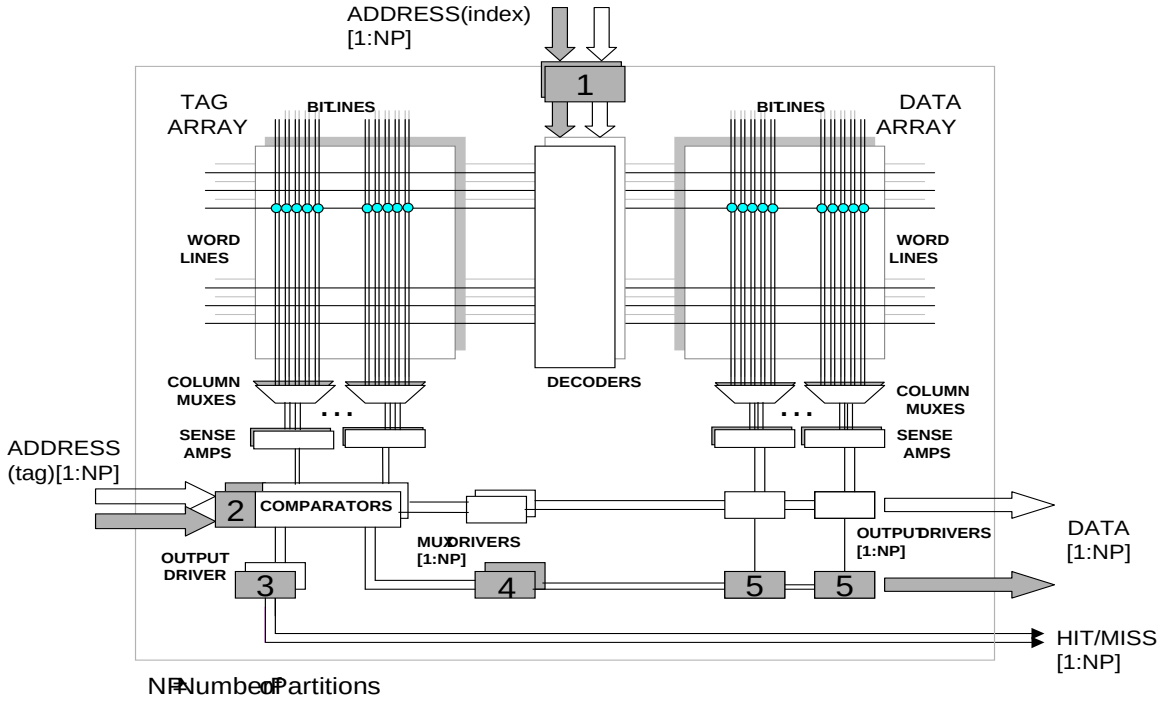


Figure 3: Implementation of reconfigurable cache. The numbered shaded blocks correspond to the modifications required over a conventional cache design.

number and dimensions of subarrays that would have been optimal for a conventional cache.

In general, a reconfigurable cache will require an equal or greater number of subarrays than a non-reconfigurable cache. An increase in the number of subarrays can have the following effects: (i) if the number of subarrays is larger, the wire area and delay required to connect them will be larger, and (ii) a larger number of subarrays results in a reduction in the number of bits in each subarray, which means the individual subarray access times are reduced. The increase in wiring delay with increasing numbers of subarrays tends to balance the decreasing delay with decreasing numbers of bits per subarray. This yields a delay curve vs. the number of subarrays that has an optimum value. The curve tends to have a fairly shallow minimum, in that configurations with similar numbers of subarrays tend to be close to the optimum. This means that a modest increase in the number of subarrays to support reconfiguration over a design point that is optimum for a non-reconfigurable design usually results in only a modest increase in delay.

One alternative to using a different organization of subarrays would have been to increase the number of ports on each subarray. However, additional ports can be very expensive: for example going from a single ported memory to a true dual-ported memory can almost double the required cache area and significantly increase its access time and power dissipation. Thus we do not further consider the provision of reconfigurability through additional cache ports in this paper.

4.3 Additional Logic for Reconfigurability

Multiplexors at the address decoders. Each address decoder needs to be preceded by a multiplexor (marked 1 in Figure 3) that selects the correct address to forward to a subarray, based on the current partitioning specified in the cache status register. The additional multiplexor delay increases the total decoder delay time. Additionally, the latency of the previous stage is increased because of the extra capacitive load imposed by the multiplexor drain capacitances. The decoder driver also needs to be duplicated at the output of each multiplexor.

Multiplexors at the tag comparators. Analogous to the multiplexers before the address decoders, multiplexors need to be added before the inputs of the tag comparators to route the tag bits from the correct input address (marked 2 in Figure 3). However, this typically does not affect the critical path of the cache access as the delay for the tag bits from the input addresses to reach the comparators is less than the delay for the tag information to arrive from the tag subarrays.

Additional multiplexor drivers and output drivers. The reconfigurable cache needs to generate multiple hit/miss signals and send back multiple data elements to the processor (one for each currently invoked partition). Therefore, the outputs of the comparators now need to drive multiple multiplexor drivers and output drivers (marked 3, 4, and 5 in Figure 3) corresponding to the various partitions.

Additional wiring. Additional wiring needs to be routed from the cache to the processor for the data paths for the

	128KB caches, 64B lines					
	8-way		4-way		2-way	
	ns	Δ	ns	Δ	ns	Δ
base	2.78	-	1.85	-	1.59	-
2-part	2.85	3%	1.92	4%	1.65	4%
4-part	2.98	7%	2.04	10%	-	-
8-part	3.20	15%	-	-	-	-
	1MB caches, 64B lines					
	8-way		4-way		2-way	
	ns	Δ	ns	Δ	ns	Δ
base	5.02	-	3.52	-	3.40	-
2-part	5.08	1%	3.56	1%	3.43	1%
4-part	5.11	2%	3.64	3%	-	-
8-part	5.34	6%	-	-	-	-

Table 2: Reconfigurable cache access times for 0.13μm technology. The *base* configuration represents the conventional non-configurable system; configurations *2-part*, *4-part*, and *8-part* represent systems with 2, 4, and 8 partitions respectively. Δ gives the access time increase over a conventional non-reconfigurable cache organization.

various partitions. This is similar to the data wires for the original cache. It will add to the complexity of routing, and modestly increase the area, delay, and power of the cache.

4.4 Impact on Cache Access Time

We use version 2.0 of the CACTI (Cache access and cycle time) model [20][1] to study the impact of the reconfigurable cache organization on the cache access time of the system.

Table 2 summarizes our results for cache sizes of 128KB and 1MB for a 0.13μm process technology (our results for other process technologies were qualitatively similar). For each cache size, three columns are shown for associativity 2, 4, and 8 respectively. For each set-associative configuration, the number of partitions is varied from 2 to the maximum number of partitions allowed (equal to the associativity of the cache), in powers of 2.

Our results show that for the various cache sizes, associativities, number of partitions, and process technologies studied, changing existing cache organizations to a reconfigurable cache organization can increase the cache access time by anywhere between 1% to 15%. For small numbers of partitions (2-way), reconfigurable caches usually increase the access time of the non-configurable baseline cache by less than 5% (4% for a 128KB cache and 1% for a 1MB cache). For larger numbers of partitions (4-way and 8-way), reconfigurable caches suffer a relatively larger cache access time penalty, particularly for smaller cache sizes (7-15% for the 128KB cache and 2-6% for the 1MB cache).

[1]The CACTI model assumes a physically-addressed cache with a two-region circuit model approximation; a more aggressive model could use a virtually addressed cache with TLB support and more complex timing models such as a four-region model [16]. However, for the first-level cache configuration we study in this paper, we do not anticipate these changes to make a qualitative difference to our results.

Sensitivity to cache size. The increase in cache access time is maximum for the smaller cache sizes. As the cache size is increased, the access time increases, and consequently, the impact of reconfigurable caches is a smaller fraction of the total access time. Also, due to the large number of subarrays usually required in larger caches, the change required to support reconfiguration is reduced.

Sensitivity to number of partitions. Across all our configurations, the cache access time increases with the number of partitions because of increased delays in driving additional loads at the multiplexors. However, with small number of partitions, the cache access time for reconfigurable caches is not significantly greater than that of non-reconfigurable designs.

4.5 Summary

Our results show that a reconfigurable cache organization can be implemented within the framework of a conventional SRAM cache design with relatively few minor changes. Our results using an analytical model of the cache access time indicate that for small number of partitions, the reconfigurable cache organization does not incur significant additional delay over traditional cache structures. Larger number of partitions have the potential to increase the cache access time, but this increase is still less than 6% with larger caches. Such an increase may or may not affect the cycle time of the machine or the number of cycles required for a cache access, depending on whether a conventional cache would limit the cycle time of the microprocessor. If the access time of a conventional cache is already somewhat below the latency of the cache (in cycles) times the cycle time of the whole microprocessor, reconfigurability could be added without negatively impacting either the machine cycle time or the number of cycles required for a cache access.

5. APPLICATION: INSTRUCTION REUSE

To provide quantitative evidence of the benefits of reconfigurable caches, we focus on one representative application of such caches. This section evaluates the performance benefits from reconfigurable caches for *instruction reuse* for media processing benchmarks. Instruction reuse has been studied in detail for SPEC benchmarks [7, 12, 13, 22, 23, 24]; however, it has not been studied in detail for media processing. We anticipate that this technique will be effective for media processing applications due to the inherent repetitiveness and incremental gradation associated with the analog data types that correspond to media data. Furthermore, previous work has showed that media applications are compute bound (versus memory bound) after the insertion of software prefetching [19]. Instruction reuse coupled with reconfigurable caches allows us to improve computation performance using otherwise underutilized memory system resources.

Section 5.1 discusses our instruction reuse implementation. Section 5.2 summarizes our simulation environment, simulated system, and benchmarks. Section 5.3 presents the performance results.

5.1 Implementation of Instruction Reuse

Originally proposed instruction reuse buffer implementation. We use an instruction reuse buffer implementation similar to that proposed by Sodani and Sohi [23] and

Processor parameters	
Processor speed	1 GHz
Issue width	8
Instruction window size	64
Functional units	
- integer arithmetic	4
- floating point	4
- address generation	4
- VIS multiplier	2
- VIS adder	2
Branch prediction	
- bimodal agree predictor size	2K
- return address stack size	32
Taken branches per cycle	1
Simultaneous speculated branches	16
Memory queue size	32

Memory hierarchy parameters	
Cache line size	64 bytes
L1 instr cache size	64KB
L1 instr cache associativity	2-way
L1 data cache size	128KB (base)
L1 data cache associativity	4-way (base)
L1 data cache request ports	4
L1 data cache hit time	2 ns
Number of L1 MSHRs	12
L2 cache size (on-chip)	1MB
L2 cache associativity	4-way
L2 request ports	1
L2 hit time (pipelined)	20 ns
Number of L2 MSHRs	12
Total contentionless memory latency for L2 misses	100 ns
Memory interleaving	4-way

Table 3: Default system parameters.

later refined by Molina et al. [17]. This scheme (referred to as S_v in [23]) detects reuse based on operand values and was shown to be the best performing scheme with large buffer sizes. In Sodani and Sohi's study, the processor is assumed to have a fixed-sized 12-ported reuse buffer and buffer sizes from 0.5KB to 12KB are considered. An instruction entry in the reuse buffer stores the source and destination operand values and part of the PC to identify the instruction. When an instruction is decoded, its source operand values are compared with those in the instruction reuse buffer entry corresponding to this instruction. If there is a match, the result is directly read from the buffer and the execution stage is bypassed. Load and store addresses as well as load values are stored in the instruction reuse buffer. To ensure consistency of data, on a store, the implementation in [23] requires a fully associative lookup of the reuse buffer for possible matching entries.

Instruction reuse buffer implementation with reconfigurable caches. When implementing an instruction reuse buffer as a partition of a reconfigurable cache, we need to make a few changes to the above design to address (i) the limited number of ports for the cache (4 cache ports vs. 12 ports in the instruction reuse buffer), (ii) the larger SRAM sizes and the inability to perform associative lookups, and (iii) increased latencies (we assume that the cache access takes 2 cycles as opposed to one cycle for a smaller dedicated instruction reuse buffer). Consequently, our implementation of the S_v scheme only stores values for arithmetic and logical instructions and addresses for memory instructions. Other instruction values (including branch outcomes and load values) are not stored in the instruction reuse buffer. This reduces the number of accesses to the instruction reuse partition. For a further reduction, on a hit in the instruction reuse buffer, we do not update the state information for replacements. The buffer is however updated on misses. For each instruction, we store as many entries (sets of values) as fit in the cache line (with 64-byte cache lines, five entries can be stored and associatively checked for each instruction).

5.2 Simulation Framework

We use the RSIM simulator [18] to evaluate the benefits from reconfigurable caches. RSIM is a user-level execution-driven simulator that models the processor pipeline and memory hierarchy in detail including contention for all resources. We model an eight-way issue out-of-order processor that includes aggressive state-of-the-art features found in several commodity microprocessors including non-blocking loads and stores, speculative execution, media ISA extensions, and software prefetching. Our simulator supports the SPARC v9 ISA and Sun's VIS media ISA extensions. Table 3 summarizes the parameters used for the processor and memory subsystems.

The base cache system uses a 128KB four-way associative first-level (L1) write-back data cache and a 1MB 4-way associative second-level (L2) write-back unified cache. The caches are non-blocking and allow support for multiple outstanding misses. At each cache, 12 miss status holding registers (MSHRs) reserve space for outstanding cache misses and combine a maximum of 8 multiple requests to the same cache line.

For the reconfigurable cache system, we partition the L1 data cache into two two-way associative partitions of 64KB each. As seen from Figure 2, the access time for this configuration is 1.92ns, 4% slower than a design without reconfigurability. One of the partitions is used as a conventional data cache while the other is used as an instruction reuse buffer (as described in Section 5.1) for the entire run of all the benchmarks.

Table 4 summarizes the benchmarks we study. The input to the benchmarks were chosen from the Intel Media Bench (described at the Intel web site) and the UCLA MediaBench suites [10]. This set of benchmarks was chosen to represent processing of a variety of media data types, including images, video, audio, and speech. The benchmarks were modified to include a call to reconfigure the caches at the beginning of the execution.

5.3 Performance Results

Figure 4 summarizes the results for all the benchmarks. For each benchmark, four bars are shown representing (i) the base system with a conventional data cache (*base*), (ii) the base system with a reconfigurable L1 data cache with two partitions – one for conventional data cache and the other for instruction reuse (*IR*), (iii) the reconfigurable cache system with "infinite" ports and partition size for the instruction reuse partition (*IRideal*), (iv) the base conventional cache

Benchmark	Description (input)
cjpeg	JPEG encoding of 1024x630 3-band image (rose16.ppm), uses VIS and software prefetching
djpeg	JPEG decoding of 1024x630 3-band image (rose16.jpg), uses VIS and software prefetching
mpegdec	MPEG2 video decoding of video stream into YUV components (meil6v2rec.m2v), uses VIS and software prefetching
mpegenc	MPEG2 video encoding of four 352x240 frames (I-B-B-P) (meil6v2rec.yuv), uses VIS
speechdec	GSM speech decoding (clinton.pcm)
audiodec	MPEG-2 audio decoding (clinton.pcm)
spchrecog	Signal cepstral feature extraction in speech recognition (ex5_c1.wav)
spchsynth	Natural language processing in speech synthesis (test_data.in)

Table 4: Benchmarks used in this study.

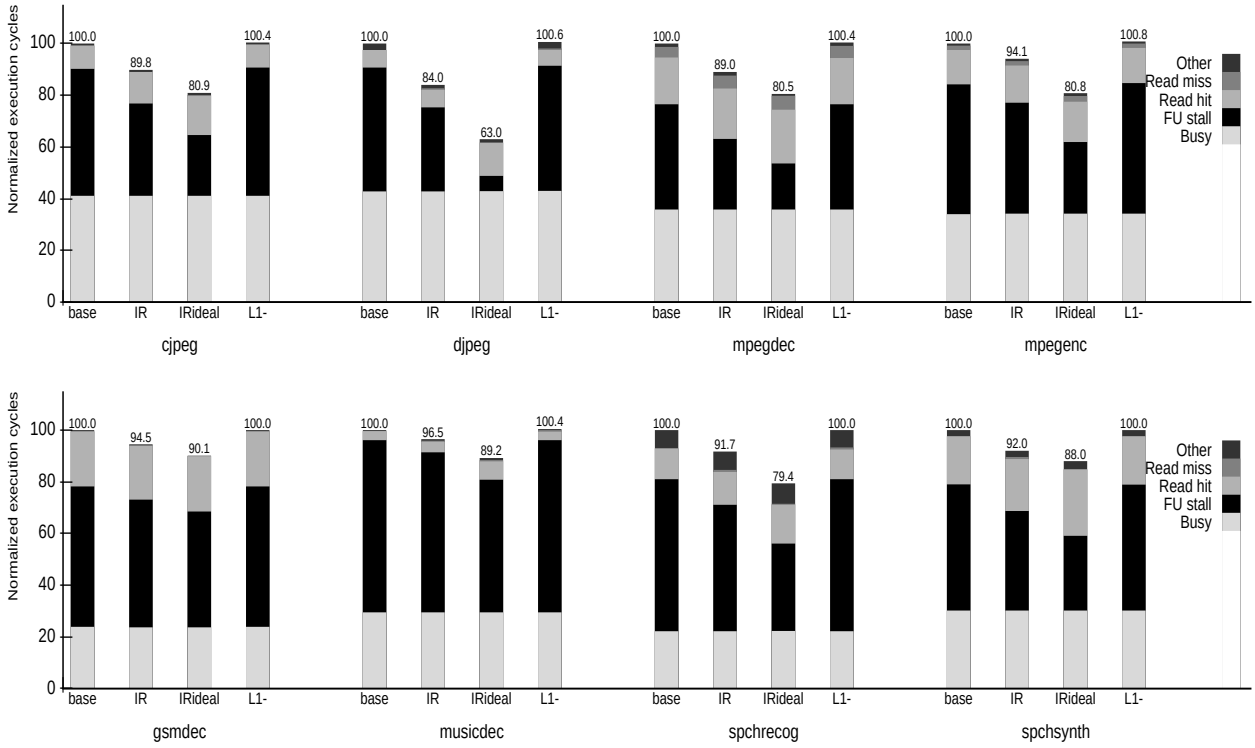


Figure 4: Simulation results for the benefits from using reconfigurable caches for instruction reuse.

system with half the L1 cache size and associativity (*L1-*); i.e., with the L1 cache available in the reconfigurable cache configuration but without the benefit of the instruction reuse partition. The bars show execution cycles (or equivalently inverse of the IPC) normalized to the cycles (inverse of IPC) of the *base* configuration. Each bar is divided further into five components – busy cycles, functional unit (FU) stall cycles, read hit stall cycles, read miss stall cycles, and other stall cycles (e.g., instruction cache miss stalls). The busy and stall cycles are calculated using the following convention, similar to that used in previous work (e.g., [19]). At every cycle, the fraction of instructions retired that cycle to the maximum retire rate is attributed to the busy time; the remaining fraction is attributed as stall time to the first instruction that could not be retired that cycle.

Comparing the *base* and *IR* configurations, we find that a realistic implementation of instruction reuse with reconfigurable caches achieves IPC improvements of 1.04X to 1.20X across our applications. The benefits are on reductions in the functional unit stall and read hit stall components of the execution cycles, enabled by bypassing the execution stage due to hits in the instruction reuse buffer.

Comparing the *base* and *IRideal* systems, the ideal instruction reuse configuration with "infinite" resources achieves IPC improvements ranging from 1.11X to 1.60X. The differences between the *IRideal* configuration and the practical implementation of the instruction reuse buffer (*IR*) stem mainly from increased port contention for the SRAM partitions used for the instruction reuse buffer (all the SRAM partitions have the same number of ports as the original SRAM cache – in our experiments, four). Enhancing the *IR* model with additional filtering mechanisms (e.g., confidence based filtering [8]) may potentially address the increased port contention and reduce the performance differences between the *IR* and *IRideal* models. However, this may involve extra hardware support; in this paper, we restrict ourselves

to reconfigurable cache organizations with marginal changes to the processor and do not study such aggressive implementations.

Finally, comparing the *base* and *L1-* results corroborates previous results that large caches are not critical for media processing workloads. Across all our applications, changing the first-level cache from a 4-way 128KB cache to a 2-way 64KB cache had marginal impact (less than 1%) on performance.

6. RELATED WORK

Albonesi has concurrently proposed the use of "selective cache ways" to selectively disable parts of the data arrays of the cache to tradeoff performance for power conservation [1]. This work also proposes to use set-associativity based partitioning and addressing of the data arrays, and cache scrubbing for data consistency when partitions are shut down. These methods are similar to those used in our primary design. Our work, however, differs from this work in several significant ways. First, in contrast to simply turning off some partitions, our work suggests using the partitions for alternate processor activities to enhance performance. This requires a more general cache design that allows multiple address inputs into and multiple data outputs out of the cache, requiring care to keep these multiple paths from contending with each other. Some of our design decisions may actually increase the power consumption in the cache arrays. Second, we perform a detailed timing analysis with the CACTI analytical model to determine the impact of our design on cache access time. Albonesi's evaluation is focused on power dissipation. Third, we evaluate the performance benefits of the reconfigurable cache organization for instruction reuse [23] for media processing. Thus, we show that cache storage can be used to improve computation speed for media processing. Albonesi focuses on shutting off parts of the cache to save power, and focuses on SPEC benchmarks.

Although we have focused mainly on reconfigurable cache organizations for general-purpose processors, they are applicable to digital signal processors as well. The recently announced Texas Instruments TMS320C62xx series of processors include support for memory systems that can be configured as cache or memory depending on the application. As discussed in Section 5, reconfigurable caches can be used for such a purpose as well. However, our reconfigurable cache organization is general enough to include other applications for the SRAM arrays including instruction reuse.

Several recent studies have examined instruction reuse; however, the key contribution of this paper is in the idea of reconfigurable caches and not instruction reuse per se. Similarly, several recent studies have proposed new architectures targeted specifically for media processing applications (e.g., [21]). The focus of our work is on designing architectures that can adaptively reuse general-purpose features for different activities for different applications, including media processing applications.

7. CONCLUSIONS AND FUTURE WORK

As general-purpose processors continue to be used for an increasing number of application domains, it is important to ensure that architectural features that use a significant fraction of the on-chip transistors are useful for most such applications. Most current general-purpose processors devote the largest fraction of the on-chip transistors to caches. However, several important applications do not exploit large caches. For example, media processing applications often find large caches to be ineffective due to their streaming data accesses and large data sets [19]. As these applications increase in importance, current general-purpose system design philosophy will force a tradeoff between resources for media processing and other application domains.

This paper proposes an alternative design using reconfigurable caches. The design evaluated in this paper provides the ability to divide the cache SRAM arrays into different partitions that can be used for different processor activities. These activities can benefit applications that could not otherwise exploit conventional caches. Our design requires very few modifications to conventional caches, exploiting the natural implementation of set-associative caches today. Detailed timing analysis using a modification of the CACTI model shows small impact on cache access time.

We suggest several applications of reconfigurable caches. To show quantitative benefits, we choose to evaluate instruction reuse for media processing as a representative application. Instruction reuse coupled with reconfigurable caches allows us to improve computation performance using otherwise underutilized memory system resources. We find IPC performance benefits of between 4% and 20% from a reconfigurable cache with two partitions, one used for conventional caching and the other for instruction reuse. Because the cache access time impact of reconfigurability is so low (4% for the organization used in the IPC simulations) it is likely that the overall microprocessor cycle time or the number of cycles required to access the cache will not be affected. Even if the microprocessor cycle time is increased by the cache access time increase, the net performance benefits still remain positive. Furthermore, comparisons with a more ideal implementation indicate that more aggressive implementations of instruction reuse can potentially achieve higher performance benefits. Additionally, using more partitions for other activities can further improve the performance benefits from this organization. It is important to note that the benefits achieved in this paper were with relatively small hardware and software changes to current general-purpose processors.

In the future, we anticipate that the paradigm of reusing on-chip storage for multiple processor activities can facilitate a number of architectural optimizations. We plan to study some of the other applications suggested in Section 2. We also plan to evaluate other options in the design space for reconfigurable caches discussed in Section 3 (e.g., more dynamic and frequent reconfiguration).

8. ACKNOWLEDGEMENTS

We would like to thank Mahesh Kallahalla, Vijay Pai, Karthick Rajamani, and the anonymous reviewers for their valuable comments and help with earlier drafts of the paper.

9. REFERENCES

[1] D. H. Albonesi. Selective Cache Ways: On-Demand Cache Resource Allocation. In *Proceedings of the 32nd*

Annual International Conference on Microarchitecture, 1999.

[2] J. M. Anderson, L. M. Berc, J. Dean, S. Ghemawat, M. R. Henzinger, S.-T. A. Leung, R. L. Sites, M. T. Vandevoorde, C. A. Waldspurger, and W. E. Weihl. Continuous Profiling: Where Have All the Cycles Gone? In *Proceedings of the Symposium on Operating System Principles*, Oct. 1997.

[3] D. Chiou, P. Jain, S. Devadas, and L. Rudolph. Application-Specific Memory Management of Embedded Systems Using Software-Controlled Caches. Technical Report CSG-Memo 427, Massachussetts Institute of Technology, November 1999.

[4] K. D. Cooper and T. J. Harvey. Compiler Controlled Memory. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 2–11, 1998.

[5] K. Diefendorff and P. K. Dubey. How Multimedia Workloads Will Change Processor Design. In *IEEE Micro*, pages 43–45, Sep 1997.

[6] J. Eyre and J. Bier. DSP Processors Hit the Mainstream. *IEEE Computer*, 1998.

[7] F. Gabbay and A. Mendelson. The Effect of Instruction Fetch Bandwidth on Value Prediction. In *Proceedings of the 25th International Symposium on Computer Architecture*, pages 272–281, 1998.

[8] D. Grunwald, A. Klauser, S. Manne, and A. Pleszkun. Confidence estimation for speculation control. In *Proceedings of the 25th Annual International Symposium on Computer Architecture*, pages 122–131, June 1998.

[9] J. Hennessy. The Future of Systems Research. *IEEE Computer*, 32(8):27–33, August 1999.

[10] C. Lee et al. MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communications Systems. In *Proceedings of the 30th Annual International Conference on Microarchitecture*, 1997.

[11] D. C. Lee, P. J. Crowley, J.-L. Baer, T. E. Anderson, and B. N. Bershad. Execution Characteristics of Desktop Applications on Windows NT. In *Proceedings of the 25th Annual International Symposium on Computer Architecture*, pages 27–38, 1998.

[12] M. H. Lipasti and J. P. Shen. Exceeding the Dataflow Limit via Value Prediction. In *Proceedings of the 29th Annual International Symposium on Microarchitecture*, pages 226–237, 1996.

[13] M. H. Lipasti, C. B. Wilkerson, and J. P. Shen. Value Locality and Load Value Prediction. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 138–147, 1996.

[14] R. Maher. Multimedia Instruction Set Extensions for a Sixth-Generation x86 Processor. In *Proceedings of HOTCHIPS-8*, 1996.

[15] A. M. G. Maynard, C. M. Donnelly, and B. R. Olszewski. Contrasting Characteristics and Cache Performance of Technical and Multi-User Commercial Workloads. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 145–156, Nov. 1994.

[16] G. W. McFarland. *CMOS Technology Scaling and Its Impact on Cache Delay.* PhD thesis, Stanford Univerisity, 1997.

[17] C. Molina, A. Gonzalez, and J. Tubella. Dynamic Removal of Redundant Computations. In *Proceedings of the ACM International Conference on Supercomputing*, June 1999.

[18] V. S. Pai, P. Ranganathan, and S. Adve. RSIM: A Simulator for Shared-Memory Multiprocessor and Uniprocessor Systems that Exploit ILP. In *Proceedings of the Third Workshop on Computer Architecture Education*, 1997.

[19] P. Ranganathan, S. Adve, and N. P. Jouppi. Performance of Image and Video Processing with General-Purpose Processors and Media ISA Extensions. In *Proceedings of the 26th International Symposium on Computer Architecture*, pages 124–135, 1999.

[20] G. Reinman and N. P. Jouppi. CACTI 2.0 Beta. In *http://www.research.digital.com/wrl/people/jouppi/CACTI.html*, 1999.

[21] S. Rixner, W. J. Dally, U. J. Kapasi, B. Khailany, A. Lopez-Lagunas, P. R. Mattson, and J. D. Owens. A Bandwidth-Efficient Architecture for Media Processing. In *Proceedings of the 31st Annual International Conference on Microarchitecture*, pages 3–13, 1998.

[22] Y. Sazeides and J. E. Smith. The Predictability of Data Values. In *Proceedings of the 30th Annual International Conference on Microarchitecture*, pages 248–258, 1997.

[23] A. Sodani and G. Sohi. Dynamic Instruction Reuse. In *Proceedings of the 24th International Symposium on Computer Architecture*, pages 194–205, 1997.

[24] A. Sodani and G. Sohi. An Empirical Analysis of Instruction Repetition. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 35–45, 1998.

[25] S. Wilton and N. P. Jouppi. CACTI: An Enhanced Cache Access and Cycle Time Model. *IEEE Journal of Solid-State Circuits*, pages 677–687, 1996.