

# Coupling Compiler-Enabled and Conventional Memory Accessing for Energy Efficiency

RAKSIT ASHOK, SAURABH CHHEDA

BlueRISC Inc.

and

CSABA ANDRAS MORITZ

University of Massachusetts, Amherst

---

This paper presents Cool-Mem, a family of memory system architectures that integrate conventional memory system mechanisms, energy-aware address translation, and compiler-enabled cache disambiguation techniques, to reduce energy consumption in general purpose architectures. The solutions provided in this paper leverage on inter-layer tradeoffs between architecture, compiler, and operating system layers. Cool-Mem achieves power reduction by statically matching memory operations with energy-efficient cache and virtual memory access mechanisms. It combines statically speculative cache access modes, a dynamic CAM based Tag-Cache used as backup for statically mispredicted accesses, different conventional multi-level associative cache organizations, embedded protection checking along all cache access mechanisms, as well as architectural organizations to reduce the power consumed by address translation in virtual memory. Because it is based on speculative static information, a superset of the predictable program information available at compile-time, our approach removes the burden of provable correctness in compiler analysis passes that extract static information. This makes Cool-Mem highly practical, applicable for large and complex applications, without having any limitations due to complexity issues in our compiler passes or the presence of precompiled static libraries. Based on extensive evaluation, for both SPEC2000 and Mediabench applications, we obtain from 6% to 19% total energy savings in the processor, with performance ranging from 1.5% degradation to 6% improvement, for the applications studied. We have also compared Cool-Mem to several prior arts and have found Cool-Mem to perform better in almost all cases.

Categories and Subject Descriptors: B.3.2 [Memory Structures]: Design Styles—*Cache Memories, Virtual Memory*

General Terms: Design, Experimentation, Performance

Additional Key Words and Phrases: Energy efficiency, Translation buffers, Virtually addressed caches

---

## 1. INTRODUCTION

The memory system is a major source of power consumption in contemporary processors. For example, the caches and translation lookahead buffers (TLB) combined consume 23% of the total power in the Alpha 21264 [Gowan et al. 1998], caches draw 42% of the energy in the StrongARM 110 [Montanaro 1997], and 23% in the PowerPC processor [Benini et al. 2000]. With the current trend of ever-increasing on-chip cache sizes, the fraction of the power consumed by caches is likely to further increase. In addition, address translation in contemporary memory systems is done with performance as the main objective, accounting for a large fraction of power consumed in the memory system. This trend fuels research to reduce power dissipation in memory systems by addressing power efficiency at all system layers: circuit, architecture, and/or software levels.

Many of the recent architecture approaches proposed, are based on the observation that not all memory system resources are required for all applications, and/or that there is a lot of resource utilization variation even within the same application. The architectural remedies proposed (e.g., in the context of caches [Benini et al. 2000; Kin et al. 1997; Huang et al. 2001; Inoue et al. 1999; Albonesi 1999; Villa et al. 2000; Balasubramonian et al. 2000]) are typically based on resizing of resources, driven by dynamic runtime information or apriori application execution profiling.

In contrast, this paper presents Cool-Mem, a family of memory system architectures, that is enabled by speculative static compile-time information. Cool-Mem integrates conventional memory access mechanism with compiler enabled techniques and energy-aware address translation, to reduce energy consumption, further blurring the interface between compiler and architecture. Our experimental results confirm our intuition, that combined compiler-architecture based designs open up smart new ways to reduce power consumption and in many cases even improve application performance. The issues raised and solutions provided in this paper leverage inter-layer tradeoffs in memory systems, clearly affecting architecture, compiler, and even operating system layers.

But how can we benefit from static information? Cool-Mem uses static program information about memory access types and patterns, to reduce some of the redundancy in conventional memory access mechanisms. This redundancy in current memory system architectures is due to the general one-size-fits-all design philosophy, where all memory accesses are treated equal, i.e., having one single dynamic approach for all situations. For example, each memory operation typically requires a TLB access for virtual-to-physical address translation or for protection checking, and every single associative cache access requires associative lookup of multiple tags and cache blocks for one single word returned. As we will show in this paper, a large fraction of this redundancy can actually be eliminated, resulting in significant power and energy savings.

Cool-Mem architectural components include: (1) support for statically speculative cache access modes, (2) a dynamic CAM based Tag-Cache used as backup for statically mispredicted accesses, (3) a conventional multi-level associative cache organization, (4) embedded protection checking along all cache access mechanisms, and (5) a variety of techniques (this because we study a number of different organizations, each with advantages and disadvantages) in supporting power-aware address translation in virtual memory architectures.

Physically tagged and indexed caches require that cache indexing be overlapped with address translation for performance reasons. As translation is overlapped with the actual cache access, only the low-order offset bits (that do not change with the translation) are available for cache indexing. With the growing on-chip cache sizes, this is becoming more and more difficult, leaving virtually tagged and indexed caches as a practical alternative [Patterson and Hennessy 1990].

Cool-Mem employs a virtually tagged and indexed cache as a first level cache and combines it with either a virtual or a physical second level cache design, with protection checks integrated along all cache access paths, including the compiler directed static access path, and moves address translation to upper layers in the

memory hierarchy to reduce the power impact of TLB accesses. Cool-Mem can also integrate with physically indexed caches; In fact, good energy savings are obtained even for this case.

Furthermore, Cool-Mem uses *speculative static* information to predict cache block translation reuse to eliminate the tag lookup for many memory accesses. For sequences of memory accesses to the same cache line, Cool-Mem saves energy by storing the cache line mapping in *Hotline registers* and using these to access that cache line directly. In the event of correct static speculation, we could simply access the cache in a more energy-efficient way. In our previous work [Unsal et al. 2001] we have shown that static speculation checking at runtime can be efficiently implemented. As opposed to other approaches to reduce energy based on predictable static information, that required program transformations such as adaptive strip-mining [Moritz et al. 1999], our approach does not affect code size and does not change the control-flow of the program. Changing control-flow could otherwise impact on the efficiency of other compiler optimization techniques and as a result degrade performance.

Additionally, a CAM based Tag-Cache complements the static access mechanisms, playing a dual role: first, it complements the static access mechanism by keeping around replaced cache mappings that were previously predicted statically, and second, it can also be used to store cache mappings for the most recently hitting cache tags for *dynamic accesses*, acting as a cache of cache-tags. Here again, energy is saved on Tag-Cache hits by directly accessing a cache line. As we will show in this paper, this mechanism is very effective in filtering out the effects of static mispredictions.

The Cool-Mem compiler extracts speculative static information, using it to match different types of accesses to different cache access modes. Because it is based on speculative static information, the burden of provable correctness in the compiler analysis is removed. This makes our approach very practical, applicable for large and complex programs. The analysis can be completed without having access to all the source codes, something that we have found to be very useful, for example, in applications with frequent calls to precompiled static libraries. Furthermore, the level of speculation can be decided at compile-time.

The main contribution of Cool-Mem is providing a hybrid power-aware memory system solution, a design where conventional hardware techniques are extended to support integration with compiler managed memory access techniques, a system that is applicable and works for large and complex programs without restrictions. To the best of our knowledge, there have been no efforts on incorporating compiler-driven statically *speculative* techniques in general purpose memory systems for power and energy savings. Based on extensive evaluation, for both SPEC2000 [Henning 2000] and Mediabench [Lee et al. 1997] applications, we obtain an energy savings from 6% to 19% in the processor. The performance obtained ranges from 1.5% degradation to 6% improvement.

The rest of this paper is structured as follows. Section 2 presents the related work. Section 2.1 provides a discussion of design challenges in virtual cache organizations. Next, we present some background on typical memory system designs in Section 3. The Cool-Mem architecture is described in Section 4 followed by Section 5 on the

compiler techniques. Section 6 details the experimental framework used, Section 7 discusses the results, Section 8 contains additional discussion, and we conclude in Section 9.

## 2. PREVIOUS WORK

Previous research focusing on TLB power consumption includes [Juan et al. 1997], which compares fully-associative, set-associative, and direct mapped TLBs from a power perspective. They also propose modifications to the basic cells and the structure of set-associative TLBs to reduce power. [Wood et al. 1986] proposes the use of large virtually tagged and indexed caches to delay the need for address translation until cache misses. Virtual to physical address translation on cache misses is done by a hardware page table walking mechanism. [Cheriton et al. 1986] details the software-controlled caching mechanism in the VMP multiprocessor. This is a TLB-less architecture with virtually addressed caches, with a software cache miss handler. Their technique however, focuses on exploring coherence in a multiprocessor. Recent papers [Jacob and Mudge 2001; 1997] also propose a virtually addressed caching architecture with software based cache miss handler, but evaluate only from a performance perspective. Our approach used in Cool-Mem in the version that is based on virtual-virtual cache hierarchies (we also consider and evaluate virtual-physical and physical-physical memory hierarchies) adds a translation buffer before accessing main memory, that reduces the overhead of address translation, improves performance, and has minimal impact on power consumption.

A sizable amount of work has been done toward improving the energy-efficiency of caches. [Kin et al. 1997] proposes a small L0 cache that saves energy when data can be found in this cache, while degrading performance by 21%. A cache way-predicting technique proposed in [Inoue et al. 1999] saves energy on correct prediction by accessing only the matching way instead of all the ways in a set-associative cache. The recent paper [Huang et al. 2001] also proposes a similar way-prediction scheme. Their cache partitioning scheme includes a specialized stack cache and compiler implementation concerns are addressed. Powell et al. [Powell et al. 2001] combine way prediction with selective direct mapping to reduce cache energy consumption. [Ma et al. 2001] proposes a deterministic way-memoization scheme as an alternative to way-prediction. Content Addressable (CAM) Tag caches are examined for low power in [Zhang and Asanovic 2000]. These techniques are purely architecture based, in contrast to our combined compiler-architecture approach.

A compiler enabled scheme, has been proposed in [Moritz et al. 2001; Moritz et al. 1999], in the context of software caches for MIT-RAW processors. Our previous work for embedded systems [Unsal et al. 2001] utilizes a compiler-managed technique in the context of embedded processors, in a tag-less single-level cache organization, using the MediaBench for evaluation. Our Cool-Mem compiler techniques expand the scope to all types of memory accesses, handle complex program structures, and support different levels of speculation. Additionally, Cool-Mem focuses on multi-level memory systems incorporating virtual memory support.

Compiler enabled techniques targeting cache energy also include the recent work in [Witchel et al. 2001]. Their scheme saves the tag-check energy when the compiler can *guarantee* an access will be to the same line as an earlier access. Their work

is similar to the predictable cache access mechanism described for Raw processors in [Moritz et al. 1999]. Their approach requires a predictable cache access to save power and it uses loop-unrolling as a way to achieve it. Additionally, their technique works only for applications with fully predictable data accesses, mainly focusing on affine array accesses and simple loop structures, and would not likely be applicable in general purpose codes. As pointed out in the introduction, our *speculative* scheme is scalable to large applications. The solution provided in Cool-Mem does not affect instruction cache performance by an increase in code-size or degrade the ability of traditional compiler optimizations.

To the best of our knowledge, this is the first comprehensive work that presents and evaluates a combined compiler-architecture scheme targeted at general purpose architectures, that extracts energy savings from the complete memory system, including both the translation hardware and the cache hierarchy.

## 2.1 Overview of Design Challenges with Virtual Cache Hierarchies

As mentioned in the introduction, physically addressed caches are becoming less and less practical with the growing cache sizes. Cool-Mem proposes to build on virtual cache hierarchies (several configurations are evaluated) that have the advantage of moving address translation to lower levels in the memory hierarchy and thus saving on power consumed for address translation (e.g., rather than doing address translation for every single memory access one would need to do it only for L1 cache misses or for L2 cache misses depending on the design).

Virtually addressed caches come with their share of problems though. The well known *synonym problem* is detailed in [Goodman 1987]. This problem arises because in a multi-programmed environment, multiple virtual addresses can map to the same physical address, to support efficient sharing/communication between processes to avoid copying, or to support more expensive message-passing models that require kernel involvement. Both hardware and software solutions to these problems have already been proposed in [Goodman 1987; Wheeler and Bershad 1992; Wang et al. 1989]. This problem is also primarily a concern in a virtual-virtual cache hierarchy and could be easily dealt in a hierarchy involving a virtually-addressed first level cache and a physically addressed second level cache [Wang et al. 1989].

Additionally, the problem can also be overcome by disallowing aliasing altogether. This can be accomplished by providing a global address space model. Other way is by forcing shared data to align in the cache, or by requiring shared data to be non-cacheable [Cheng 1987]. Furthermore, other solutions allow arbitrary aliases but implement a consistency protocol in hardware [Wang et al. 1989].

Other hardware based techniques to deal with the synonym problem are based on ideas such as back pointers [Wang et al. 1989] or dual tag sets (having both a physical and a virtual tag, as in the ARM10 for example), or reverse translation tables [Smith 1982] that translate physical addresses into virtual addresses. Reverse address translation can possibly also use a similar approach to the virtual-physical translation and leverage hardware support for translation buffers similar to TLBs.

As mentioned earlier, a software based solution to solve or avoid aliasing is also possible, based on setting operating system policy. If energy-efficiency is an important design constraint, this approach together with other techniques that avoid dealing with aliasing are to be preferred. The IBM OS2 operating system for exam-

ple, places all shared segments at identical virtual addresses in all process address spaces. SunOS uses a different approach; it aligns shared pages on large virtual boundaries, making sure that aliases map to the same cache block [Chen et al. 1992]. Single address space operating systems that use global addresses would not have to deal with aliasing. Example of such systems include Opal [Chase et al. 1992] and Psyche [Scott et al. 1988]. Such systems eliminate the need for virtual-address aliasing by having all shared data through global references, allowing pointers to be shared freely.

A solution that has been used to deal with the problem of integrating physical IO into virtual memory hierarchy is to use reverse translation tables; a natural place to put that table is at the second level. Depending on how frequently this translation is used, it will impact power consumption however. An important issue is how to deal with this problem in bus based multiprocessor architectures based on snooping, when the second level cache is also virtual, and this translation is done frequently. One solution to that problem is to use both physical and virtual addresses on the bus, or only virtual addresses if single global address space support is provided in the operating system.

Another issue that can be raised, if a large second virtual level cache is used, is the impact on the context-switch time. The concern is that if all the cache blocks are required to be flushed and (some) written back to the next level in the memory hierarchy, this may cause a significant overhead. As shown, however, in the works of Goodman [Goodman and Woest 1988] and Wang [Wang et al. 1989], this effect is negligible for smaller caches such as L1, and can be handled efficiently for larger caches. The idea is to use an additional bit per tag that is set during context-switch. This bit together with the dirty and the valid bits can be used during block replacements to distribute write-backs (require the use of write-buffers) in time (i.e., they will happen during actual replacement and not once during context-switch time), thus avoiding cost of the latency associated with write-backs. Additionally, having ASIDs (Address Space Identifiers) in the tag bits enables smooth transition between processes. DMA accesses can be supported by flushing affected cache blocks before the transfer or having regions in the memory that are non-cacheable.

### 3. BACKGROUND

Contemporary microprocessors have complex memory systems that differ from each other in the way caches are accessed, TLB-misses are handled, etc. Nonetheless, the defining features remain the same. To model these in our baseline architecture, we have chosen an Alpha-like [Compaq 1995] memory system architecture, with physically tagged and indexed caches. Figure 1 shows the components of this architecture. Each memory instruction has 3 operands: the destination register *r1*, base register *r2*, and an immediate offset. The base and offset are used to calculate the effective address, shown as step 1 in the figure. The generated virtual address is 64-bits, which consists of the virtual page number and offset, and an unused part that sign-extends the virtual address. This virtual address is first translated into a physical address by the TLB (step 2). Each access is associated with a 7-bit Address Space Identifier (ASID) which is also fed to the TLB to check access rights.

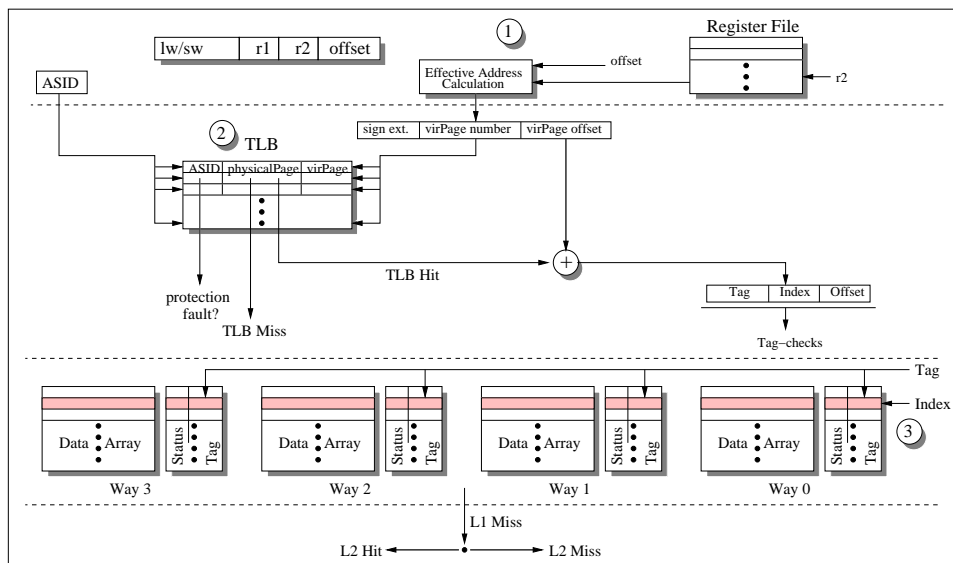


Fig. 1. The baseline memory system. All accesses require a TLB access and tag-checks.

TLB-misses are handled by a hardware page-table walking mechanism.

The cache index component of the virtual address is used to index into one of the cache sets. This enables the 4 cache lines and associated tags in that set of the 4-way cache (this can be extrapolated for higher associativity caches). The tag component of the virtual address is compared with the 4 cache-tags in parallel (step 3). At the same time as these tags are being compared, the 4 cache lines are accessed in parallel (step 3), and the cache-line offset is used to index the required word in the lines. Depending on which of the cache-tags match, if any, one of these words is selected and the access is satisfied. A miss in the L1 cache goes to the L2 cache which is similarly accessed.

The three energy-consuming components in an L1 access are: (1) fully-associative TLB access, (2) 4 parallel tag-checks, and (3) 4 parallel cache line accesses. We target these components to extract energy savings in Cool-Mem.

#### 4. COOL-MEM ARCHITECTURE

Conventional general-purpose microprocessors use a one-size-fits-all access mechanism for all accesses. The Cool-Mem architecture derives its energy savings by providing different energy-efficient access paths that are compiler-matched to different types of accesses. We proceed by providing an overview of the Cool-Mem architecture and follow with detailed discussions on the key features of this architecture.

We propose and evaluate several organizations. In both organizations we use a virtually-indexed and virtually-tagged first level cache and move address translation to lower levels in the memory hierarchy (nevertheless, the Cool-Mem architecture is general enough to be applied to physically-tagged cache hierarchies, as

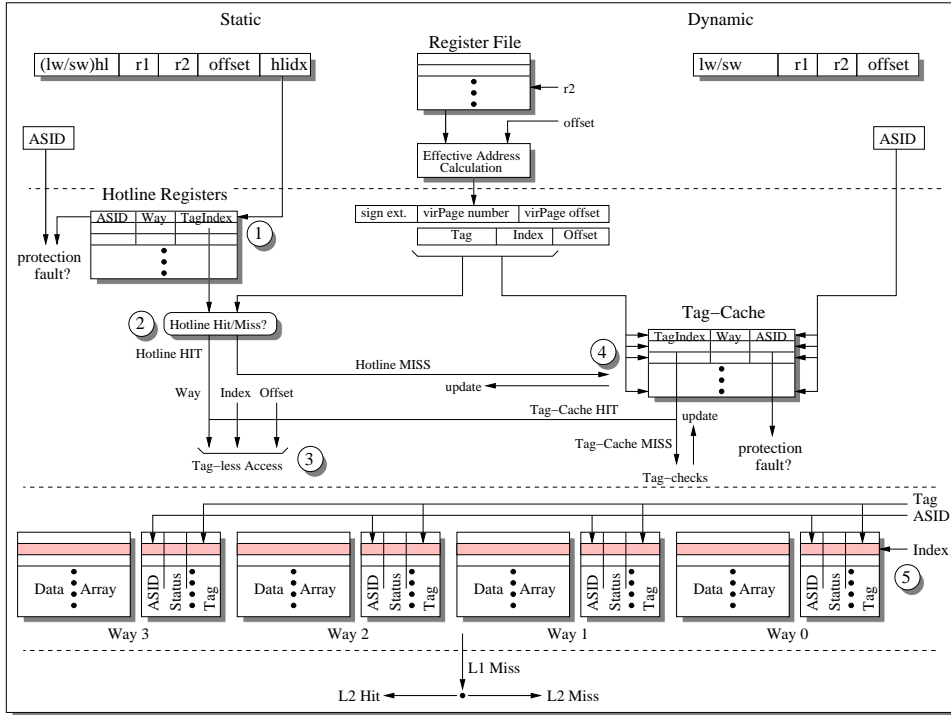


Fig. 2. The Cool-Mem memory system. Static accesses require an extra hlidx field. Notice the absence of a TLB on L1. The ASID bits on each hotline register, Tag-Cache entry, and each cache-line in L1 and L2 caches provide protection checks. The hotline-check occurs at an earlier stage than the Tag-Cache access.

will be demonstrated in the results section). As second level, we evaluate both a physically-indexed and a virtually-indexed cache. As described in section 2.1 some of the design challenges in virtual-virtual organizations (e.g., the synonym problem, integration in bus based multiprocessor systems, and context-switching with large virtual L2s) could be handled easier in virtual-physical designs. In both organizations, we add translation buffers. In the virtual-virtual (v-v) organization, a translation buffer (MTLB) is added after the L2 cache and is accessed for every L2 cache miss. We found this to serve better our energy optimization objectives than a TLB-less design, where address translation is implemented in software. Nevertheless, if maximum flexibility is desired in the way paging is implemented in the operating system, the TLB-less design is a reasonable option, as shown by our experimental results. In the virtual-physical organization (v-r), a translation buffer (STLB) is added after the L1 cache and is accessed for every L1 cache miss or every L2 cache access.

An overview of the different cache organizations with address translation moved toward lower levels in the cache hierarchy is shown in Figure 3. As address translation consumes a significant fraction of the energy consumed in the memory system,

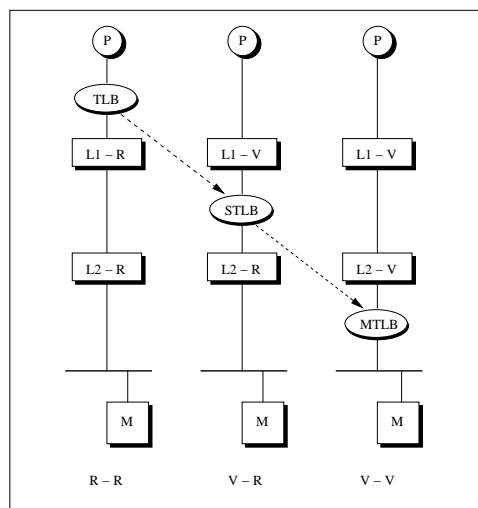


Fig. 3. Various cache organizations, with address translation moved toward lower levels in the memory hierarchy. Cool-Mem is based on the virtual-virtual and virtual-physical designs. STLB is the translation buffer between L1 and L2, and MTLB is the translation buffer added between L2 and main memory.

both the v-v and v-r designs will save energy compared to a physical-physical (r-r) cache hierarchy, where virtual-to-physical address translation is done for every memory access. Although TLB-less designs have been suggested in v-v type of organizations before, we are not aware of any proposal where address translation is done with translation buffers for L2 cache misses, as opposed to implemented in software exception handlers. Similarly, v-r designs have been applied recently (e.g., StrongARM SA-1100 processor), these designs typically do the address translation in parallel with the L1 cache access. In addition, we are not aware of any work evaluating the power-performance tradeoffs due to address translation in these organizations and in a design integrated with compiler managed access paths. A context-switch between threads belonging to different tasks may require change in virtual address mappings. To avoid flushing the TLBs we added address-space identifiers to TLB entries. Note that not having the address-space identifiers not only would require flushing all the TLB entries, but would also imply that the newly scheduled thread, once it starts executing, will experience a number of TLB misses until its working set is mapped.

Figure 2 presents an overview of the Cool-Mem memory system, with integrated static and dynamic access paths. Cool-Mem extends the conventional associative cache lookup mechanism with simpler, direct addressing modes, in a virtually tagged and indexed cache organization. This direct addressing mechanism eliminates the associative tag-checks and data-array accesses. The compiler-managed speculative direct addressing mechanism uses the *hotline registers*. Static mispredictions are directed to the CAM based *Tag-Cache*, a structure storing cache line addresses for the most recently accessed cache lines. Tag-Cache hits also di-

rectly address the cache, and the conventional associative lookup mechanism is used only on Tag-Cache misses. Integration of protection-checks along all cache access paths enables moving address translation to lower levels in the memory hierarchy or TLB-less operation. In case of TLB-less designs, an L2 cache miss requires virtual-to-physical address translation for accessing the main memory; a software virtual memory exception handler can do the needful.

#### 4.1 Support for moving the TLB to lower levels in the memory hierarchy or TLB-less operation

Cool-Mem focuses on virtually addressed caches, and integrates support for protection checks, otherwise performed by the TLB, along all access mechanisms. That is, Cool-Mem has embedded protection checks in the Hotline registers, the Tag-Cache, and cache tags. Cool-Mem therefore could completely dispense with the TLB. L2 cache misses in the v-v organization require address translation for the main memory access. Cool-Mem uses translation buffer to speed up this address translation, but a software VM exception handler for doing the translation on L2 cache misses and fetching the data from the main memory can also be used. This would be a possible source of performance overhead; however, if the L2 miss rate is much smaller compared to the TLB miss rate (in our baseline r-r architecture), there may even be some performance improvement. Additionally, Cool-Mem can be integrated with physical-physical(r-r) memory hierarchies, as is shown in the results section.

#### 4.2 Hotline Registers

The conventional associative lookup approach requires 4 parallel tag-checks and data-array accesses (in a 4-way cache). Depending on the matching tag, one of the 4 cache lines is selected and the rest discarded. Now for sequences of accesses mapping to the same cache line, the conventional mechanism is highly redundant: the same cache line and tag match on each access. Cool-Mem reduces this redundancy by identifying at compile-time, accesses *likely* to lie in the same cache line, and mapping them *speculatively* through one of the hotline registers (step 1 in Figure 2). As shown in [Cortadella and Llberia 1992] the condition that the hotline path evaluates, can be done very efficiently without carry propagation. This enables the hotline check to be started before the effective address calculation. The hotline cache access can also be started in parallel with the check, with the consequence that in case of incorrect prediction some additional power is consumed in the data-array decoder. This power is included in our experimental results using Wattch. As a result, the primary source of latency for hotline based accesses, is due to the data array access and the delay through the sense amps. Note that conventional associative cache designs require an additional multiplexer stage to select between ways in a multi-way access.

Furthermore, as shown in [Reinman and Jouppi 1999], the critical path is typically the tag-path; the tag latency can be as much as 30% larger than the latency of the data-array path in the conventional design. As shown in [Iyer and Marculescu 2001], reduced feature sizes in next generation architectures will further accentuate the latency increase of the tag path. Because of this, in conventional cache designs, the way-selection logic is moved toward the tag to balance the delay differences

between the tag and data-array paths [Reinman and Jouppi 1999]. However, in Cool-Mem the latency of the data-array could be the main target for optimizations, as the tag path is not on the critical path for most of the memory accesses, by adequate bitline and wordline partitioning. Additionally, as physical cache designs would require the TLB access completed to perform the tag comparison (the tag access could be however done in parallel), this may also add to the tag path latency. As such, we expect that a Cool-Mem based microprocessor could either have a faster clock or at least a faster cache access for statically predicted cache accesses.

The different hotline compiler techniques are described in Section 5. A simple run-time comparison (step 2) reveals if the static prediction is correct. The cache is directly accessed on correct predictions (step 3), and the hotline register updated with the new information on mispredictions. We have included a fully associative lookup on the hotline registers to support invalidations.

As shown in Figure 2, the hotline register has 3 components: (1) protection bits (ASID), which are used to enforce address space protection, (2) TagIndex - two accesses are to the same cache line if their Tag and Index components are the same. The TagIndex component is compared with Tag and Index of the actual access to check if the hotline register can indeed be used to directly address the cache, (3) cache-way information - this information enables direct access to one of the ways in the set-associative cache.

### 4.3 Tag-Cache

Another energy-efficient cache access path in Cool-Mem is the CAM-based Tag-Cache. It is used both for static mispredictions (hotline misses) and accesses not mapped through the hotline registers, i.e., *dynamic accesses* (step 4). Hence it serves the dual-role of complementing the compiler-mapped static accesses by storing cache-line addresses recently replaced from the hotline registers, and also saving cache energy for dynamic accesses; the cache is directly accessed on Tag-Cache hits (step 3).

The motivation behind using the Tag-Cache as a backup mechanism for hotline registers is illustrated by the example in Figure 4. Due to the irregular nature of  $a[b[i]]$  accesses, it is unknown at compile-time if successive accesses are spatially close, i.e., likely to map to the same cache line, and therefore should be hotlined. The Cool-Mem compiler may aggressively map  $a[b[i]]$  through a hotline register. Depending on the values of  $b[i]$  at run-time, the access pattern for  $a[b[i]]$  may look as in Figure 4. The figure shows  $a[b[i]]$  mapping to the same cache line  $c_1$  for some period. These accesses are obviously hotline-hits. Following this period, we have an access pattern such that successive accesses map alternately to two cache lines,  $c_2$  and  $c_3$  respectively. The first access misses both the hotline and the Tag-Cache, and hence the hotline register and Tag-Cache are updated with pointers to  $c_2$ . Next access maps to  $c_3$  - again the hotline register and Tag-Cache are updated with  $c_3$ . The key point here is that the  $c_2$  pointer is *replaced* with the  $c_3$  pointer in the hotline register, whereas the Tag-Cache contains both  $c_2$  and  $c_3$  mappings. The following access to cache line  $c_2$  is therefore a hotline-miss but will be captured by the Tag-Cache. In fact, because of the alternating pattern, all further accesses will be hotline-misses, but are captured by the Tag-Cache. In general, the Tag-Cache

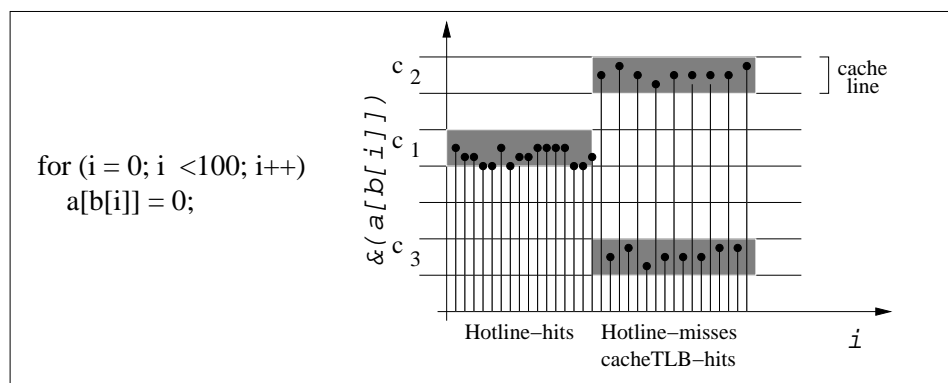


Fig. 4. Hotline-misses hitting the Tag-Cache. We found that aggressively speculative hotline accesses that cause hotline-misses, hit the Tag-Cache with good probability.

acts as a very good backup because it contains previously predicted hotline entries and does a fully associative search on these.

Although the Tag-cache access is very quick, we assume (conservatively) that the Tag-Cache, accessed on hotline misses, requires another cycle, with an overall latency similar to a regular cache access in an r-r organization. A miss in the Tag-Cache implies that we fall back to the conventional associative lookup mechanism with an additional cycle performance overhead (step 5). The Tag-Cache is also updated with the new information on misses, in LRU fashion. As seen in Figure 2, each Tag-Cache entry is exactly the same as a hotline register, and performs the same functions, but dynamically.

#### 4.4 Associative Lookup

The Cool-Mem associative cache lookup is slightly different from the conventional lookup in that the protection information (ASID) is also tagged to each cache line. Even the virtually addressed L2 cache is tagged with protection information in the v-v design to enable TLB-less L2 access. This increases the area occupied by the tag-arrays, and also its power consumption. Compared to the overall cache area and energy consumption, this increase is negligible.

### 5. COOL-MEM COMPILER

We have utilized the SUIF infrastructure for our compiler framework. The Cool-Mem compiler is responsible for identifying groups of accesses *likely* to map to the same cache-line, and mapping them through one of the hotline registers. This *hotline pass* expands on our previous work [Moritz et al. 1999; Moritz et al. 2001; Unsal et al. 2001], adding support for various levels of speculation and leveraging type information to enlarge its scope to all types of memory accesses. As opposed to our previous work, we do not use alias analysis. We found that for large applications such as those in SPEC2000, a flow-sensitive and context-sensitive analysis is not practical due to complexity issues, static library calls, and because of complex program constructs such as pointer based calls and recursive procedures found in

many of these programs. Rather, we restricted ourself to compiler analysis that would make our system applicable to all type of applications, without limitations. We are currently pursuing research in the direction of a speculative flow-sensitive alias analysis that can be adapted to further complement the techniques we describe here.

Some accesses are fairly regular and the compiler can map these through the hotlines quite accurately. Other types of accesses are very hard to analyze at compile-time. For such accesses, the compiler is not sure whether to map them statically through the hotline registers. This scenario motivates us to implement various levels of speculation in the hotline pass. Specifically, we have implemented two hotline passes: (1) Optimistic Hotlines, where the compiler tries to map *all* accesses through the hotline registers, and (2) Conservative Hotlines, which maps a *subset* of the accesses that are more regular in nature and as a result, are likely to cause fewer mispredictions. We now present these two compiler techniques.

### 5.1 Optimistic Hotlines

The goal of hotlines analysis is to identify memory accesses with high reuse and map it to one of the hotline registers. There are four types of accesses that we identify as having high reuse. These are accesses that have:

- (1) Temporal reuse, for example, across loop iterations.
- (2) Self-spatial reuse across iterations:

```
for(i=0; i<100; i++) A[i] = ...
```

- (3) Group-spatial, for example, accesses  $A[i]$  and  $A[i+2]$  in a loop body
- (4) Simple-reuse, e.g., a pointer that points to various parts of small data structures that may have high cache line reuse

The hotline pass works on a per procedure basis, see Algorithm 1. The input to this algorithm is the set of hotline registers,  $\{r_1, r_2, \dots, r_h\}$ , and the control flow graph (CFG) of the function. It parses through the CFG, mapping *each* access through one of the hotline registers. To decide which of the  $h$  hotline registers to map an access through, it compares this access with all the  $h$  previous accesses currently mapped through the  $h$  hotline registers, and finds the one *spatially closest* to this access. If the distance between these is small compared to the cache line-size, they are very likely to lie in the same cache line, and therefore the current access is mapped through the same hotline register as this closest access. Otherwise, the least recently used hotline register is picked, and the current access is mapped through this register. We chose the *threshold distance* when two accesses are mapped to the same hotline as half the cache line size.

In evaluating the distance between two accesses, the hotlines pass leverages control-flow, loop structure, and type information: field offsets in structures, array element sizes, etc. Finding the distance between two accesses has running time  $O(1)$ . For each access, the hotlines pass has to compute  $h$  such distances. Therefore, if a procedure has  $n$  memory accesses, the hotline algorithm has running time  $O(nh)$ . We now illustrate the working of this algorithm with some simple examples.

Figure 5(a) shows an example with array accesses within a loop that have constant index differences. Suppose the array element-size is 4 bytes and the cache

(a)	<pre> for (i = 0; i &lt; 100; i++)   a[i]{1} = a[i+1]{1} + a[i+100]{2} + a[i+103]{2}; </pre>	<pre> // Affine array access with // constant index differences </pre>
(b)	<pre> for (i = 0; i &lt; 100; i++)   a[i]{..} = a[i]{..}*var1.field1{1} + a[i+1]{..}*var1.field2{1} + a[i+2]{..}*var1.field10{2} </pre>	<pre> // Non-array accesses </pre>

Fig. 5. (a) Example with affine array accesses, (b) Example with non-array accesses. The numbers in curly brackets are the hotline registers assigned by the hotline pass

line is 64 bytes, implying a threshold distance of 32 bytes. The hotline analysis first assigns  $a[i]$  hotline register  $r_1$ . When it comes to  $a[i+1]$ , it checks the distance from currently mapped accesses, and finds the closest one to be  $a[i]$  which is 4 bytes apart. Since this is less than the threshold,  $a[i+1]$  is also mapped through  $r_1$ . Similarly for  $a[i+100]$ , the closest access,  $a[i+1]$ , is found to be 396 bytes apart, and hence  $a[i+100]$  is mapped through a different hotline register  $r_2$ . Because  $a[i+103]$  is within the threshold distance from  $a[i+100]$ , it is mapped through  $r_2$ .

The Cool-Mem compiler technique is not limited to arrays. Figure 5(b) shows how non-array accesses are treated. Suppose `var1.field1` has been assigned  $r_1$ . If the field offsets for `field1`, `field2`, and `field10` in the structure variable `var1` are 0, 4, and 40 respectively, the hotline pass will map `field2` through  $r_1$  (distance 4 bytes from `field1`  $<$  threshold) and `field10` through  $r_2$  (distance 36 bytes from `field2`  $>$  threshold).

## 5.2 Conservative Hotlines

This flavor of the hotline pass is more selective in mapping accesses through hotline registers. It works in almost the same manner as the Optimistic Hotlines algorithm, and also has a running time of  $O(nh)$ . The key difference is the logic for selecting which accesses to map through hotline registers.

Our experiments have revealed that pointer-based accesses typically have low static prediction rates, especially without the information provided by a precise alias analysis pass. Non-affine array accesses of the form  $a[b[i]]$  are also a prime source of mispredictions. The Conservative Hotlines pass does *not* map these two types of accesses through the hotlines. This conservative approach thus maps fewer accesses through the hotline registers than the Optimistic scheme, but hopes to achieve better prediction rates for these accesses.

Figure 6(a) gives an example with non-affine array accesses. Due to the irregular nature of this access, it has the potential for causing many mispredictions. The Conservative Hotlines algorithm does not hotline this access. If  $b[i]$  turns out to be very regular, for example  $b[i] = i$ , then hotlining this access makes sense. This is however very unlikely, and the Conservative Hotlines algorithm achieves a better prediction rate than the optimistic counterpart in most cases.

Figure 6(b) presents an example with pointer-based accesses. The linked-list structure referenced by  $p$  is also unpredictable. Dynamic allocation, insertions and deletions mean that the memory layout of the list is very irregular. Hence, it would be unwise to assume that successive nodes in the list are spatially close. The

<pre> /* Irregular array accesses */ for (i = 0; i &lt; 100; i++)   a[b[i]]{not mapped} = 0; </pre> <p style="text-align: center;">(a)</p>	<pre> /* Pointer-based accesses */ for (i = 0; i &lt; 100; i++) {   p-&gt;val{x} = a[i]{..};   p{x} = p-&gt;next{x}; } </pre> <p style="text-align: center;">(b)</p>
--	--

Fig. 6. (a) Example with irregular array accesses, (b) Example with pointer-based accesses. An “x” in the curly brackets means that the access is not hotlined.

conservative approach chooses not to hotline pointer-based accesses at all.

## 6. EXPERIMENTAL FRAMEWORK

### 6.1 Compiler

We have used the SUIF/MachsuiF suite as our compiler infrastructure. Figure 7 traces the steps involved in going from the source code to alpha binary code. The source files are first compiled into suif code and merged into one file. All the high-level compiler analysis passes, including the hotline pass, operate at this stage. The hotline pass assigns hotline registers to memory accesses by *annotating* them. The annotations are propagated to the binary file through the intermediate stages. These *Alpha* binaries are simulated on the SimpleScalar [Burger and Austin 1997] simulator with all the required modifications in place.

### 6.2 Simulator

We have used the SimpleScalar [Burger and Austin 1997] simulator with Wattch [Brooks et al. 2000] extensions for collecting performance and energy numbers. This simulator, capable of running statically linked alpha binaries, has been modified to accommodate the Cool-Mem architecture. It also has the modification required to recognize the annotated load/store instructions as hotline accesses.

### 6.3 Benchmarks

We have used the CPU2000 [Henning 2000] and Media-bench [Lee et al. 1997] applications for evaluation purposes. Six Mediabench and six CPU2000 benchmarks have been randomly chosen, see Table I. To keep the excessively large simulation time for the CPU2000 benchmarks within manageable limits, we skip the first 500 million instructions and simulate the next 1 billion instructions, similar to [Sair and Charney 2000].

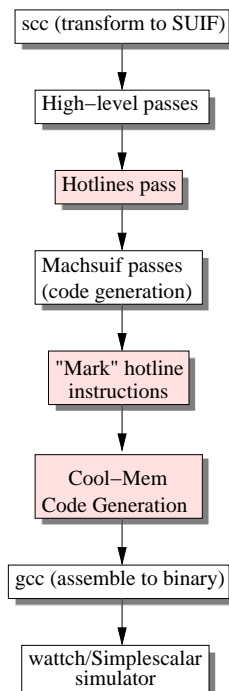


Fig. 7. Shaded steps are the ones introduced by Cool-Mem.

Table I. Benchmarks used for evaluation

Benchmark	Description
ADPCM	Adaptive Differential Pulse Code Modulation coder (Mediabench)
EPIC	Image compression utility based on wavelet decomposition (Mediabench)
G721	Voice compression based on G.711, G.721 and G.723 standards (Mediabench)
JPEG	A lossy image compression decoder (Mediabench)
MPEG	Lossy motion video compression decoder (Mediabench)
RASTA	Speech recognition front-end processing (Mediabench)
MCF	Combinatorial optimization, single-depot vehicle scheduling (CPUInt00)
PARSER	Word processing, synthetic English parser (CPUInt00)
VPR	CAD FPGA circuit placement and routing (CPUInt00)
AMMP	Computational chemistry (CPUFP00)
ART	Image recognition using neural networks (CPUFP00)
EQUAKE	Simulation of seismic wave propagation in large valleys (CPUFP00)

Table II. Baseline System Parameters

Parameter	Baseline value
Processor Speed	1.5Ghz
Process technology	0.18 $\mu$ m, 2V
Hotline registers	32
TAG-Cache size	32-entry
Fetch/Issue/Decode width	4-way out-of-order
Integer units	4
Floating-point units	4
ITLB	64-entry Fully-assoc.
DTLB	64-entry Fully-assoc., dual ported
TLB-miss penalty	20 cycles
L1 D-cache	64k, 4-way, 64byte line, dual ported
L1 I-cache	64k, 4-way, 64byte line
Unified L2 cache	512k, 4-way, 128b line
L1 D-Cache latency	3 cycles
L2 latency	20 cycles
Main memory latency	200 cycles + 2cycles/word

Table III. Power consumption breakdown for the L1 D-cache in Cool-Mem

Hardware Block	“ON” Power Consumption
32 Hotline registers	0.108211W
32-entry Tag-Cache	1.0237W
Associative Data-array access	9.59838W
Associative Tag-array access	1.53072W

## 7. RESULTS

In this section, we compare the Cool-Mem family with the baseline architecture. In these experiments, we have accounted for the energy consumed by all the added hardware blocks and any slowdown incurred. The baseline system configuration is shown in Table II and Table III shows the power consumption breakdown for the L1 D-Cache in Cool-Mem.

Table IV shows the power consumption breakdown for the various processor components. These static figures are assuming “maximum power” operation, i.e., all

Table IV. Processor power consumption breakdown

Hardware Block	
I-TLB	0.99%
D-TLB	1.99%
L1 I-Cache	6.37%
L1 D-Cache	12.7%
L2 Unified Cache	5.41%
Branch Pred.	5.27%
Register File	4.16%
Integer ALU	5.42%
Floating-point ALU	16.6%
Clock Power	34.2%
Other	6.81%

the units are fully active on all the ports with maximum switching activity. During execution though, the memory hierarchy (TLBs and Caches) may be more stressed than the floating point unit, for some applications, say. The power consumption due to the memory system will account for a higher fraction than that reported in this table, for such cases. Also note from the table that the clock dissipation accounts for a seemingly unreasonably large portion. One reason is that Wattch somewhat overestimates the clock energy by considering a model that is uniform across the whole chip. Unfortunately, a fully accurate model would require a more detailed low-level analysis of clock power, which is beyond the scope of this paper. Further, the clock power, as modeled by Wattch, accounts for all clock capacitance, including clock nodes within individual units, i.e., power dissipation in clock nodes that are internal to various hardware structures is counted towards the global clock power rather than the particular hardware units.

The hotlined load/store instructions require an additional 5 bits (for a Hotline register file size of 32). We have tried to quantify the effect of this by running experiments with one extra instruction per basic block. Figure 8 shows the performance degradation due to this for different fetch/issue/decode widths. Our results show that the code dilution can be kept below 5% if only critical blocks are optimized. For example, more than 90% of the runtime has been captured with less than 35% of the instructions. That is, the dilution is only affecting only 35% of the code. As seen in the figure, the runtime performance overhead is minimal if there are at least two functional units. In a 2-way issue processor the overhead was less

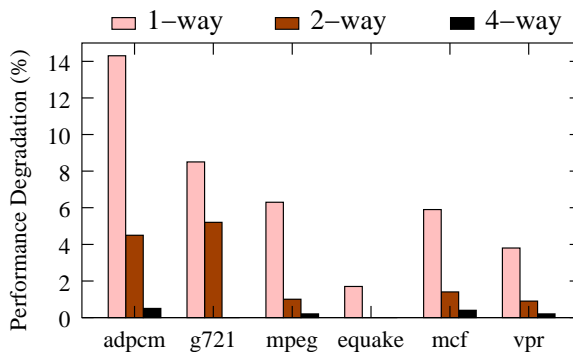


Fig. 8. Performance degradation for 1, 2, and 4-way issue processors.

than 5% and in a 4-way issue like the one evaluated in our experiments, less than 1% for the applications studied. Note that in a real implementation, that overhead could be optimized across other optimizations than those applied in Cool-Mem, e.g., similar compiler-enabled techniques for the I-cache and perhaps fetch throttling [Unsal et al. 2002].

In the following sections, we first show Cool-Mem results with identical system configuration as the baseline (see Table II), for both the virtual-real Cool-Mem architecture (v-r) and the virtual-virtual Cool-Mem architecture (v-v) (overall numbers are presented for a real-real Cool-Mem architecture as well). Next, we present sensitivity results by changing certain baseline parameters, i.e., Cache line size, Hotlines register-file size, TAG-Cache size, Cache size, Optimistic vs Conservative Hotlines, and hardware TLB vs software managed address translation.

### 7.1 Baseline Cool-Mem Results

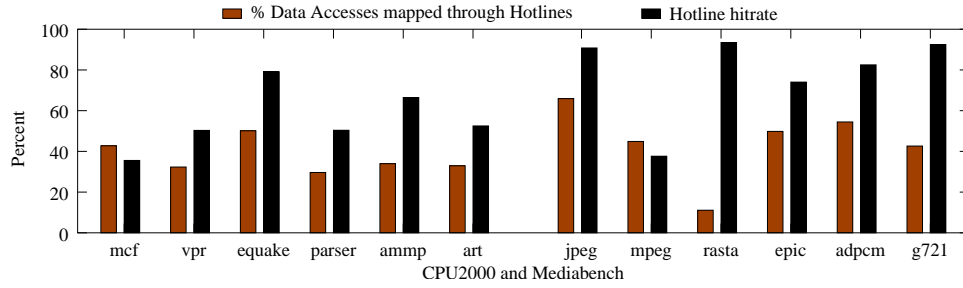
The difference between v-r and v-v designs is after the L1 cache layer (TLBs are between L1 and L2 in v-r and after L2 in v-v), and therefore, the L1 D-Cache is accessed exactly the same way in both v-r and v-v. Hence, the energy and performance numbers for the Hotlines, the TAG-Cache, and the L1 caches are same for both v-r and v-v. These results are discussed in the following paragraphs.

Figure 9(a) shows the percentage of accesses that are *hotlined* and the hit rate on these accesses. On average, 37% of the accesses for CPU2000, and 45% of the accesses for MediaBench are hotlined. The remaining accesses, i.e., *dynamic* accesses are caused primarily by library calls, the source code of which is unavailable during the hotlines analysis stage. For example, “rasta” makes heavy use of the math library calls, and the non-library memory operations are thus a small fraction of the total. As to the hit rates, 56% of the static speculations in CPU2000 and 79% in MediaBench, turn out correct on average.

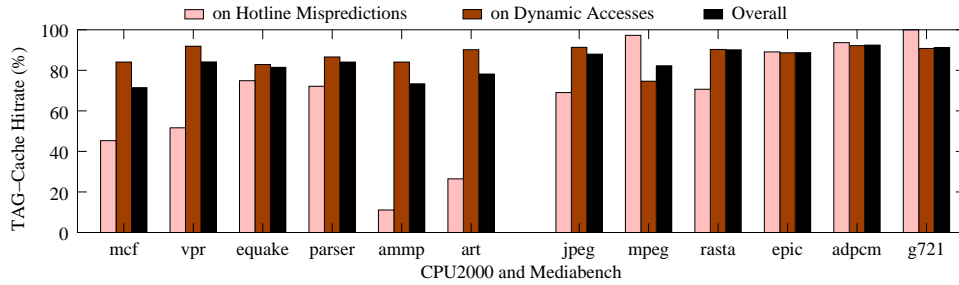
The performance penalty due to the mispredictions is diluted by the backup mechanism: the TAG-Cache. Shown in Figure 9(b), the TAG-Cache absorbs 47% of the mispredictions in CPU2000 and 87% in Mediabench. Further, as will be shown, the static hit rate for CPU2000 can be improved with Conservative Hotlines analysis. Figure 9(b) also shows the TAG-Cache hit rate on dynamic accesses: this averages at 87% for CPU2000 and 88% for MediaBench applications. The overall hit rate is 79% and 89% for CPU2000 and MediaBench applications, respectively.

Figure 9(c) shows the L1 D-Cache access patterns for CPU2000 and MediaBench applications. The lower bars are the 2-cycle static hits, the middle bars are 3-cycle TAG-Cache hits, and the upper bars are 4-cycle TAG-Cache misses that are L1 hits, and the rest are L1 misses. Figure 9(d) shows the relative energy consumption in the L1 D-Cache, broken down into various components. 100% corresponds to the L1 D-cache energy consumption in the baseline architecture. The average relative consumption value is 38% for CPU2000 (or 62% cache energy savings) and 30% for MediaBench (70% savings).

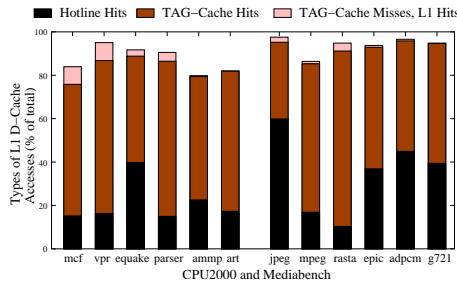
We now move beyond the L1 cache and present overall energy and performance results. Here we have also presented experimental results for a Cool-Mem architecture based on a physically-tagged cache hierarchy (r-r). In Figure 10(a), we show the performance gains in r-r, v-r, and v-v. The r-r design results in performance ranging from 0.44% degradation to 5.72% improvement. The difference in perfor-



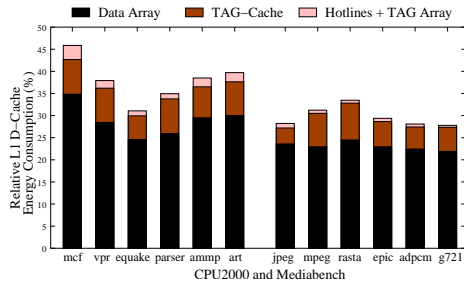
(a)



(b)



(c)

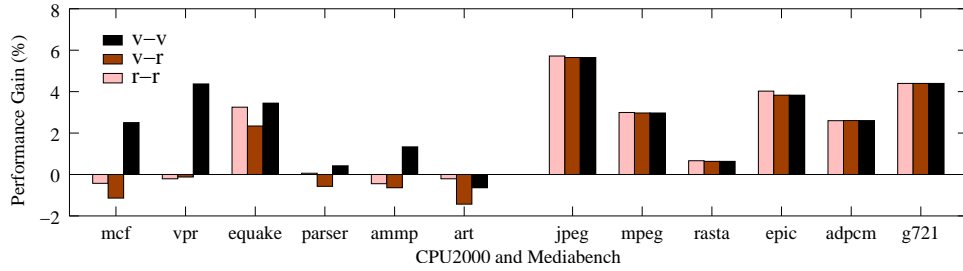


(d)

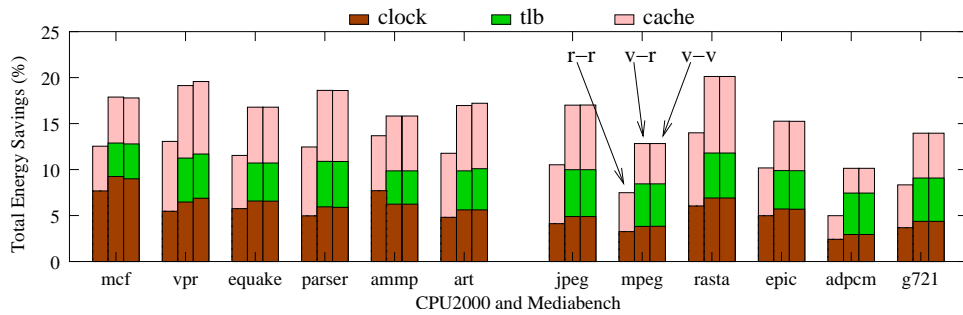
Fig. 9. Baseline results

mance, compared to the baseline, is due to the faster hotlines access mechanism as well as the overhead on TAG-Cache misses. For most applications, the performance benefits due to hotline hits outweighs the overhead due to TAG-Cache misses, and an overall improvement is observed. Average performance gains of 0.38% for CPU2000 and 3.4% for MediaBench are achieved.

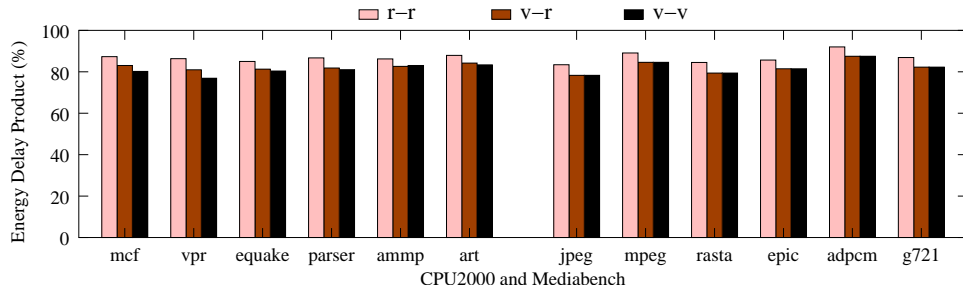
For the v-r design, we get performance ranging from 1.43% degradation to 5.65% improvement. On average, a miniscule 0.26% performance degradation for CPU2000



(a)



(b)



(c)

Fig. 10. Baseline Results

and 3.35% improvement for MediaBench is achieved. Compared to the r-r design, a small degradation observed for some applications is because the TLB is now on the L2 access path, making L2 accesses more expensive by 1 cycle.

The v-v design has higher performance gains: 1.91% for CPU2000 and 3.35% for MediaBench applications on average, with an overall range of -0.64% to 5.65%. The better figures for v-v are because there are fewer TLB misses in v-v (TLB accessed on L2 misses in v-v as against on L1 misses in v-r). For v-v, the TLB is on the main

memory access path, making these accesses costlier by a cycle. An application with sufficiently high L2 miss rate can get a degradation because of this penalty (e.g. “art”).

Figure 10(b) shows the total energy savings for the three designs. A portion of the energy savings is due to reduction in clock energy consumption: as energy consumption in the caches is reduced, the clock energy consumption also comes down because clock energy is directly proportional to chip-activity. For the r-r design, the savings are coming from the L1 D-cache mainly whereas in v-r and v-v designs, energy savings are also obtained in the TLBs. Excellent energy savings are obtained, averaging 13% for CPU2000 and 10% for MediaBench in the r-r design and the corresponding numbers being 18% and 15%, in both v-r and v-v designs.

The Energy-delay product is plotted in Figure 10(c). For CPU2000, the average energy-delay products are 80.8% for r-r, 86.6% for v-r, and 82.3% for v-v. These numbers follow the performance trend: on average for CPU2000, r-r has the best performance, followed by v-v and v-r. For MediaBench, average energy-delay products are 82.3% for r-r, 86.9% for v-r, and 82.3% for v-v. Again r-r and v-v have better results than v-r due to better performance numbers.

## 7.2 Sensitivity Analysis

In this subsection, we evaluate the sensitivity of Cool-Mem to various architectural and compiler parameters.

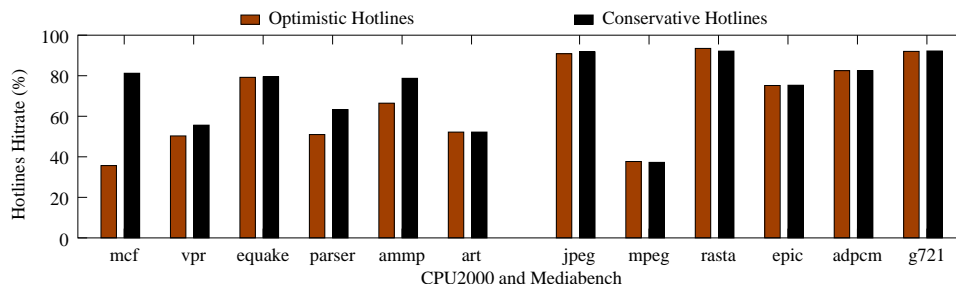


Fig. 11. Varying levels of Speculation in the Cool-Mem Compiler

**7.2.1 Optimistic Vs Conservative Hotlines.** The motivation behind having Optimistic and Conservative versions of the hotline analysis was to filter out the less regular accesses and not map these through the Hotline registers in the conservative scheme. This, we hoped, would lead to an improvement in the Hotline hit rate. Figure 11 shows the Hotline hit rates for the two versions. The Conservative approach *does* improve the hit rate for CPU2000 applications by 13% on average. The Mediabench applications perform almost the same in both cases. This is because these media applications have very regular access patterns; there aren’t many irregular accesses that the conservative approach can filter out.

A few of the applications which had low static hit rates with the Optimistic technique don’t gain much from the Conservative technique (e.g. “vpr”, “art”,

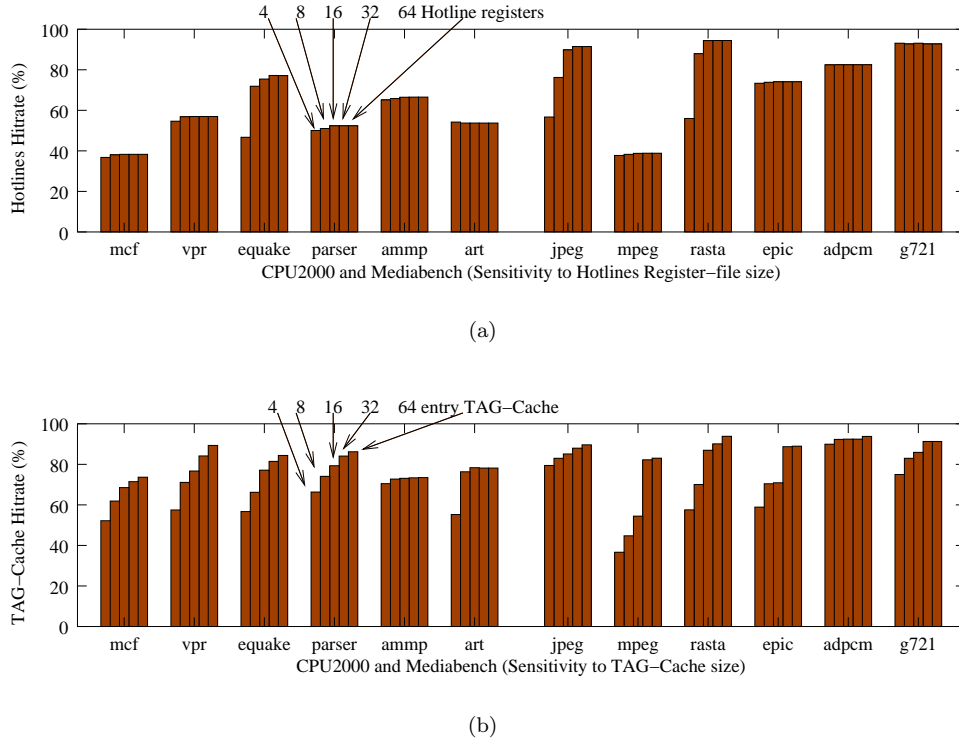
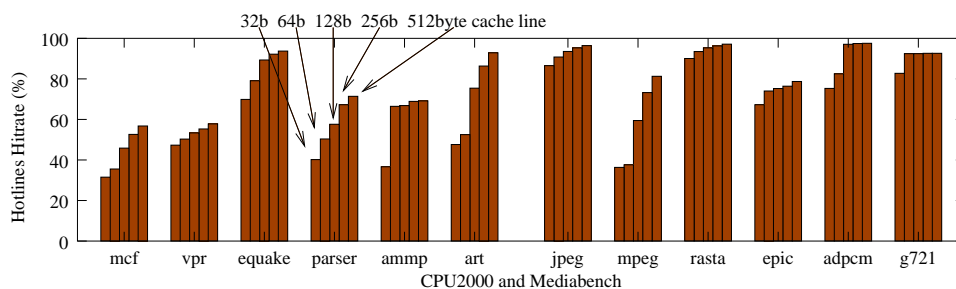


Fig. 12. Sensitivity to Hotline register-file and TAG-Cache sizes

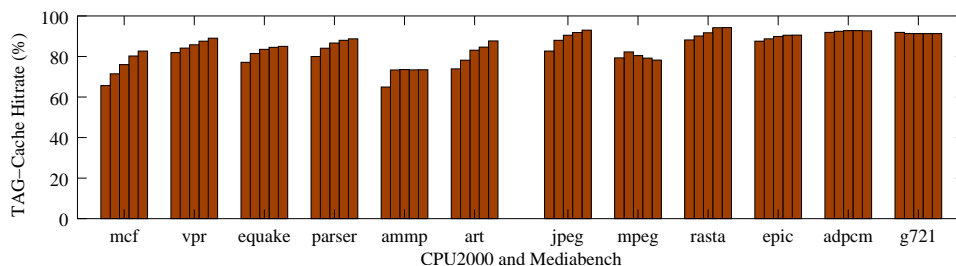
“mpeg”). This is because some of the affine array accesses hotlined by the Conservative algorithm have very big strides, and therefore should not be hotlined. But the compiler may be unable to determine the stride at compile to decide whether or not to hotline this access. An even more conservative approach would be not to hotline even affine array accesses, when the stride is not known at compile time.

**7.2.2 Hotline Register-file and TAG-Cache sizes.** Figure 12(a) looks at static hit rate with increasing number of hotline registers. For smaller number of hotline registers, there is a higher change of conflicts, i.e. two spatially far-apart accesses being assigned the same register. As expected, we see an improvement in the hit rate with increasing number of hotline registers. Saturation occurs around 16 registers, with minor improvement going to 32 registers. This suggests that most of the innermost loops of these programs have at most about 10 hotlined accesses.

Figure 12(b) shows the TAG-Cache hit rates for different sized TAG-Caches. The “epic” and “mpeg” benchmarks see a big improvement when the TAG-Cache size increases from 16 to 32 entries. For other applications, there is a very gradual improvement with increasing TAG-Cache size, with saturation occurring at a TAG-Cache size of 32 entries.



(a)

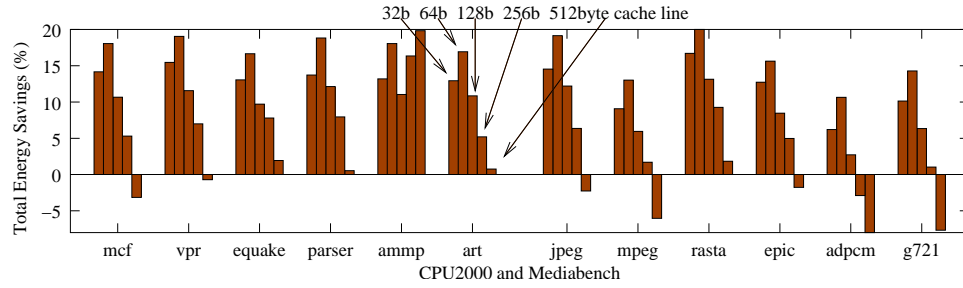


(b)

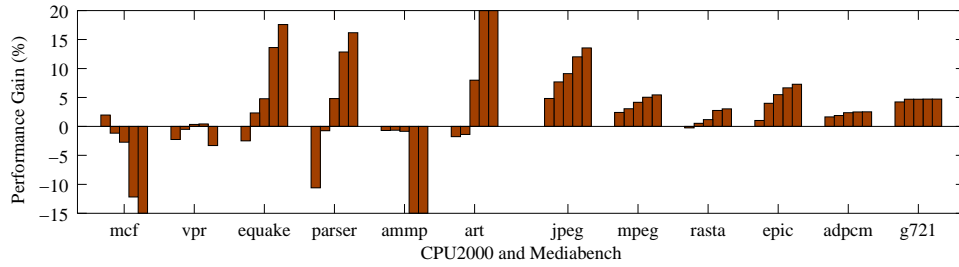
Fig. 13. Sensitivity to Cache line size

**7.2.3 Cache Line Size.** Figure 13(a) shows how the variation in Hotline hit-rate with increasing L1 D-Cache line sizes. The static hit rate is seen to be fairly sensitive to cache line size: As line size increases, the Hotline hit-rate is also seen to increase. This is expected, because the likelihood of an access mapping to a particular cache line increases as the line size increases. The same argument should also hold for Tag-Cache hit rates. Indeed, the Tag-Cache hit rate also increases with increasing cache line size, as shown in Figure 13(b). However, there is a second-order effect of increasing line size: as the Hotlines hit-rate increases with increasing line size, the remaining Hotline mispredictions become more and more irregular; i.e., the TAG-Cache hit-rate on the Hotline mispredictions goes down with increasing line size. This second-order effect may sometimes manifest itself in an overall lowering of the TAG-Cache hit-rate with increasing line size (i.e., "mpeg" in Figure 13(b)).

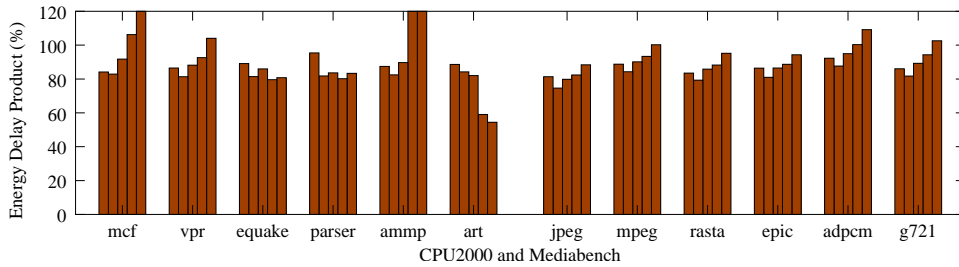
Figure 14(b) shows performance gain for v-r with increasing line sizes. Most of the applications show improved performance with bigger line sizes. This is because these applications exhibit considerable spatial locality: the L1 hit rate goes up when cache line size is increased from 32bytes to 512bytes. Another reason for the improved performance is better Hotline and TAG-Cache hit rates with bigger line sizes. Exceptions to the trend are "mcf", "vpr", and "ammp". These applications don't show as much spatially locality as the others: L1 miss rate goes up with increasing line sizes, resulting in performance degradation.



(a)



(b)



(c)

Fig. 14. Sensitivity to Cache line size

Figure 14(a) shows the variation in energy savings with increasing line sizes for v-r. Bigger lines causes more energy dissipation in the Data-Arrays of caches, meaning that cache energy savings vanish very quickly with increasing line size. This is the reason for the general downward trend in energy savings. Such applications as “mcf” and “vpr”, suffer from higher L1 miss rates with larger cache lines, as stated in the last paragraph. They incur further energy penalty because of this, as L2 accesses consume more energy than L1 accesses, and thus the sharp diminishing of energy savings for these applications.

Table V. Sensitivity to Associativity: Delay

Benchmark	4-way			2-way		
	r-r	v-r	v-v	r-r	v-r	v-v
mcf	1.00	1.01	0.97	0.99	1.02	0.98
vpr	1.00	1.00	0.96	0.99	1.00	0.96
equake	0.97	0.98	0.97	0.97	0.98	0.97
parser	1.00	1.01	1.00	1.00	1.01	1.00
ammp	1.00	1.01	0.99	1.00	1.01	0.99
art	1.00	1.01	1.01	1.00	1.01	1.00
jpeg	0.94	0.94	0.94	0.94	0.94	0.94
mpeg	0.97	0.97	0.97	0.97	0.97	0.97
rasta	0.99	0.99	0.99	0.95	1.00	0.99
epic	0.96	0.96	0.96	0.96	0.96	0.96
adpcm	0.97	0.97	0.97	0.97	0.97	0.97
g721	0.96	0.96	0.96	0.96	0.96	0.96

Table VI. Sensitivity to Associativity: Energy

Benchmark	4-way			2-way		
	r-r	v-r	v-v	r-r	v-r	v-v
mcf	0.87	0.82	0.82	0.86	0.86	0.86
vpr	0.86	0.81	0.80	0.84	0.84	0.84
equake	0.88	0.83	0.83	0.87	0.86	0.86
parser	0.87	0.81	0.81	0.84	0.85	0.85
ammp	0.86	0.82	0.84	0.86	0.86	0.88
art	0.88	0.83	0.83	0.86	0.86	0.86
jpeg	0.89	0.83	0.83	0.86	0.86	0.86
mpeg	0.92	0.87	0.87	0.90	0.89	0.89
rasta	0.85	0.80	0.80	0.83	0.84	0.84
epic	0.89	0.85	0.85	0.88	0.87	0.87
adpcm	0.94	0.90	0.90	0.93	0.91	0.91
g721	0.91	0.86	0.86	0.89	0.88	0.88

Figure 14(c) gives the sensitivity of the energy delay product for increasing line sizes. A line size of 64bytes generally gives the best result.

**7.2.4 Cache Associativity.** Table V shows the relative performance for the r-r, v-r, and v-v Cool-Mem designs based on 4-way and 2-way associative caches. Performance values are scaled relative to the baseline for the given cache associativity; values lesser than 1 indicate performance gain and values greater than 1, an overhead (for example, a value of 0.5 means the application completes in only half the cycles, etc.). Similarly, energy savings and the energy-delay product are shown in Tables VI and VII, respectively.

**7.2.5 Cache Size.** Figure 15 shows the percent of static hits, TAG-Cache hits, L1 hits and L1 misses for three different cache sizes. The static and TAG-Cache hit rate is seen to be almost independent of cache size. This is because the hotline registers and the TAG-Cache entries store information for the most recently used 32 cache blocks, and as long as the cache has more than 32 blocks, the hit rates will stay the same. Obviously, the L1 miss rate goes down with increasing cache size.

Table VII. Sensitivity to Associativity: Energy-Delay Product

Benchmark	4-way			2-way		
	r-r	v-r	v-v	r-r	v-r	v-v
mcf	0.87	0.83	0.80	0.85	0.87	0.83
vpr	0.86	0.81	0.77	0.84	0.85	0.80
equake	0.85	0.81	0.80	0.84	0.84	0.83
parser	0.87	0.82	0.81	0.84	0.85	0.84
ampp	0.86	0.83	0.83	0.86	0.86	0.86
art	0.88	0.84	0.83	0.86	0.87	0.86
jpeg	0.83	0.78	0.78	0.81	0.81	0.81
mpeg	0.89	0.85	0.85	0.87	0.87	0.86
rasta	0.85	0.79	0.79	0.79	0.83	0.83
epic	0.86	0.81	0.81	0.84	0.84	0.84
adpcm	0.92	0.88	0.88	0.90	0.89	0.89
g721	0.87	0.82	0.82	0.85	0.84	0.84

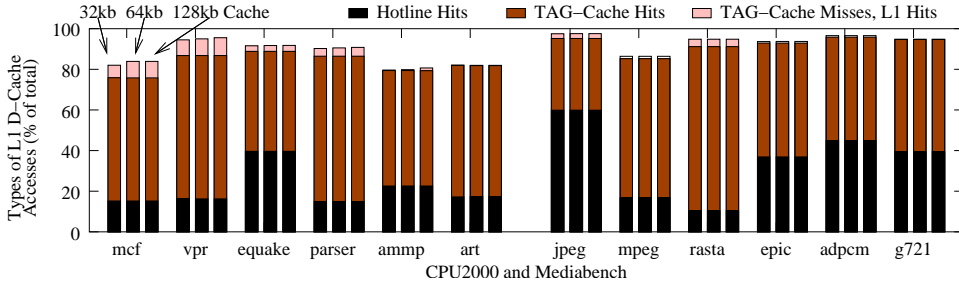


Fig. 15. Hotlines and TAG-Cache sensitivity to cache size

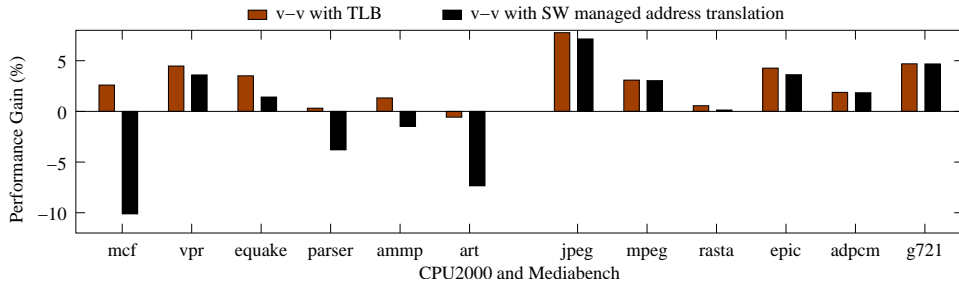


Fig. 16. v-v performance: with TLBs vs with software managed address translation

7.2.6 *TLB vs Software Managed Address Translation.* Figure 16 shows the performance improvement for v-v over the baseline, for two cases: (1) TLBs are present, and (2) address translation is managed by software. We have charged 20 cycles as the average time for software-based address translation (which is reasonable, given that software-based TLB miss handler was 20 cycles). Software-managed address translation is more flexible than the TLB solution, for example, problems associ-

ACM Journal Name, Vol. V, No. N, Month 20YY.

ated with virtual addressing can be effectively taken care of here. The price for this flexibility is performance. Still, even with software-based address translation, performance is gained on all MediaBench applications and half of the CPU2000 applications.

### 7.3 Comparison to Prior Art

We have implemented three hardware based techniques from prior work to compare Cool-Mem against:

- (1) **Last cache line TAG buffer:** Here the TAG corresponding to the last cache line accessed is stored in buffer/register. In case of a hit, direct access to one of the cache ways is effected. In terms of energy-savings, this is similar to a hotline hit in the Cool-Mem r-r design. Since the last cache line TAG check is similar to a hotline check, and can be performed in the earlier pipeline stages, we have accounted for a 1-cycle saving on hits, and no degradation on misses.
- (2) **Filter Cache:** This is an implementation of the Filter Cache [Kin et al. 1997]. The Filter Cache parameters are the ones resulting in best overall performance, as reported in [Kin et al. 1997]: 256byte, Direct-Mapped, 32byte line. On Filter-Cache hits, energy-savings even greater than those corresponding to Hotline hits in Cool-Mem are obtained due to the small L0 cache size. The L0 access time is also lower than that for the L1: we account for a 1-cycle performance gain on L0 hits.
- (3) **Hardware Way-Prediction:** This is an implementation of the work proposed in [Inoue et al. 1999]. Here the last way accessed in each set is stored in an array: on correct predictions, direct access to one of the cache-ways results in energy savings comparable to that on Hotline hits in r-r Cool-Mem.

In the following paragraphs, we compare the three Cool-Mem designs with the three aforementioned hardware based schemes, on the three metrics: performance, energy-savings, and energy-delay product.

Figure 17(a) compares the performance gains of last cache line TAG buffer(1TAG), Filter Cache(Filter), and Hardware Way-prediction, with the three Cool-Mem designs(r-r, v-r, v-v). 1TAG has minimal performance gain for all the applications(due to low hit-rate). However, no performance degradation is observed for any of the applications(because misses have no overhead). Filter Cache suffers from severe performance degradation due to very low L0 hits rates (around 65-70%). Hardware Way-Prediction carries a 1-cycle overhead on mispredictions(similar to TAG-Cache misses in Cool-Mem). However, due to the very good hit rates, hardly any performance degradation is observed. The three Cool-Mem designs do better than these hardware based schemes across the board.

Figure 17(b) compares the energy-savings. Due to the additional ITLB and DTLB savings in Cool-Mem v-r and v-v designs, they again do better than all the other schemes across the board. It is interesting to compare Cool-Mem r-r(which doesn't have any TLB energy savings) with the three hardware schemes. Due to the lower hit-rates in 1TAG and Filter, they perform worse than r-r. However, Filter does *better* than all other schemes for the memory bound application *mcf*. Even with the not-as-good hit-rate as Cool-Mem, due to the extremely small L0

size, the energy savings are greater. Hardware way-prediction has pretty good rates and does slightly worse or slightly better than r-r.

Figure 17(c) compares the energy-delay product(lower is better). Again, v-r and v-v do better than the rest, across the board. Due to the lower energy-savings in 1TAG, and much worse performance in Filter, these two schemes have worse results compared to Cool-Mem r-r. Energy-wise Hardware way-prediction performs on par with r-r, but doesn't have the performance gain of r-r(due to hotline hits in r-r): Cool-Mem r-r is the winner here for most of the applications.

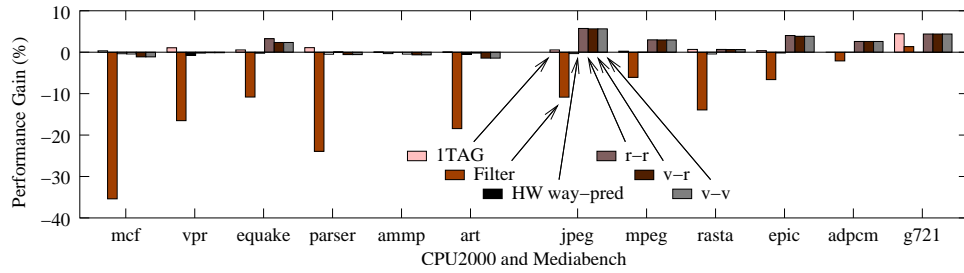
In summary, we observe that Cool-Mem v-r and v-v designs do better than the sampling of prior art compared against, for all the applications tested on. Even the Cool-Mem r-r design, which doesn't have any ITLB/DTLB energy-savings, does better, with only Hardware way-prediction coming close.

## 8. ADDITIONAL DISCUSSION

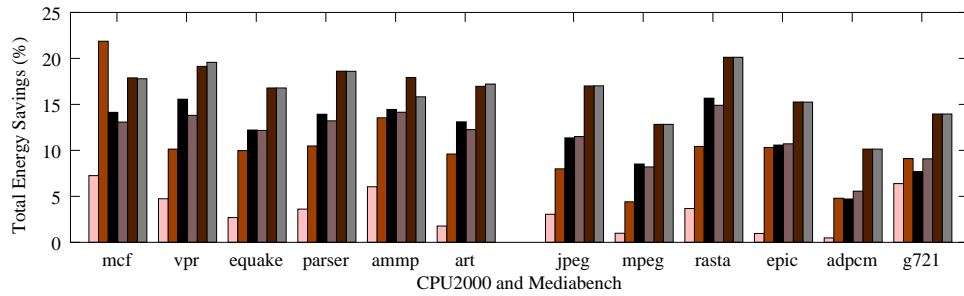
Cool-Mem techniques can be used even in the case of CAM-Tag based caches. Many low-power embedded microprocessor's, e.g., ARM10, ARM11, XScale, implement CAM tags. ARM cores can also implement Ram-Tag based caches especially when they are synthesized from soft-core versions. While CAM-based caches serialize the tag access, they have the advantage of making higher associativity and cache-line locking easier to implement. Cam-Tagging has been shown recently to result in minimal penalty on cache access and area [Zhang and Asanovic 2000] compared to Tag-Ram based approaches, for smaller size L1 caches (e.g., 2K to 16K). Typical embedded microprocessor caches would bank the cache such that each bank contains sets from all ways. Thus, the size of the bank becomes a factor that determines/limits the associativity of the design.

Cool-Mem techniques could be applied to determine which bank and way is used in a CAM-Tagged cache. The hotline registers would in that case contain the way and bank information. The Cool-Mem compiler would pretty much be unchanged. A memory operation that would access the cache through the hotlines path would have its cache *match* line directly enabled and would avoid accessing the CAM-Tags. Thus, while the Cool-Mem techniques in a Ram-Tag design save energy by removing Tag lookups and redundant associative data-array accesses, in the case of CAM-Tag caches, the saving would be entirely from eliminating Tag accesses. Note however that in CAM-Tag caches the CAM access is the most significant energy component due to the highly associative lookup and comparison that is implemented for each way in a cache CAM bank. Depending on the size of the cache, the energy saving in removing a Tag lookup, in a typical CAM-Tag cache, is around 65-75% [Zhang and Asanovic 2000].

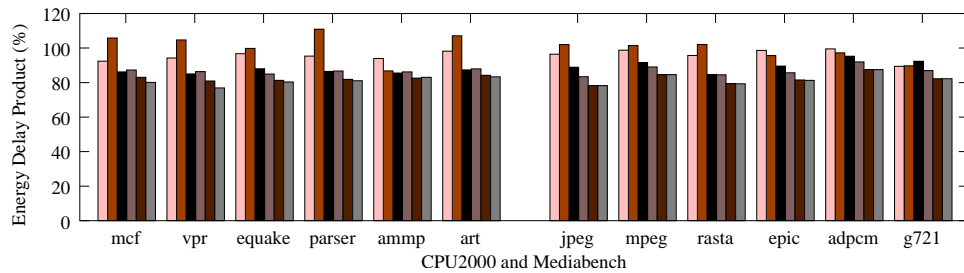
Several circuit-level techniques are currently being proposed to address leakage power in next generation process technologies [Chandrakasan et al. 2000; Montanaro 1997; Kao and Chandrakasan 2000; Kuroda et al. 1998; Shigematsu 1997; Borkar et al. 1998; Mutoh et al. 1995; Kuroda and Sakurai 1996; Hu et al. 2002]. While leakage power is expected to be significant in future deep-submicron technology nodes (especially if assuming current implementation approaches), we expect that circuit-level techniques will go a long way to reduce it. Nevertheless, compiler-architecture based leakage reduction could be easily integrated into Cool-Mem, and



(a)



(b)



(c)

Fig. 17. Comparison Results

we are currently pursuing research in that direction. For example, in a CAM-based design one could use the bank information to selectively precharge the accessed bank only, saving on the bitline energy. Furthermore, a miss in the Hotline registers could be used as an indicator for when a cache-line will likely not be requested anymore (i.e., due to the fact that affine accesses in hotlines could be assumed to access the cache in a monotonic way), and used to move the old cache-line into a low leakage state, saving cell leakage. Clearly, a leakage-aware cache cell design such as the drowsy cache [Flautner et al. 2002] or  $V_{DD}$ -gated [Powell et al. 2000] cache is

needed to enable it.

## 9. CONCLUSION

This paper described Cool-Mem, a novel memory system architecture based on tight integration between compiler and architecture, that combines conventional memory system mechanisms, selective address translation, with compiler-enabled statically speculative memory accessing techniques, to reduce energy consumption in general purpose architectures. The issues raised and solutions provided in this paper leverage inter-layer tradeoffs in memory systems, clearly affecting architecture, compiler, and even operating system layers. Cool-Mem achieves significant energy reduction in the processor, ranging from 6% to 19%, with performance ranging from 1.5% degradation to 6% improvement, by statically matching memory operations with energy-efficient cache and virtual memory access mechanisms. Cool-Mem makes several contributions: (1) it shows how to integrate statically speculative mechanisms in general-purpose memory systems; (2) describes a practical compiler framework where static speculation can be controlled with different analysis and required architectural support; (3) successfully designs architectural backup mechanisms to work together with compiler-enabled ones; and (4) provides architectural support for selective address translation including support in static access paths. This paper also provides a detailed analysis of both compiler-level and architectural design points, includes a number of sensitivity analysis to the design parameters selected, that we believe will make it possible to incorporate Cool-Mem in next generation general purpose microprocessors.

## REFERENCES

- ALBONESI, D. H. 1999. Selective cache ways: On-demand cache resource allocation. In *International Symposium on Microarchitecture*.
- BALASUBRAMONIAN, R., ALBONESI, D. H., BUYUKTOSUNOGLU, A., AND DWARKADAS, S. 2000. Memory hierarchy reconfiguration for energy and performance in general-purpose processor architectures. In *International Symposium on Microarchitecture*.
- BENINI, L., MACHI, A., AND PONCINO, M. July 2000. A Recursive Algorithm for Low-Power Memory Partitioning. In *Proceedings of the International Symposium on Low Power Electronics and Design (ISPLED '00)*.
- BORKAR, S., YE, Y., AND DE, V. 1998. A Technique for Standby Leakage Reduction in High-Performance Circuits. In *Symposium on VLSI Circuits, pages 40-41*.
- BROOKS, D., TIWARI, V., AND MARTONOSI, M. June 2000. Wattch: a framework for architectural-level power analysis and optimizations. In *Proceedings of the 27th International Symposium on Computer Architecture (ISCA '00)*.
- BURGER, D. C. AND AUSTIN, T. M. 1997. The SimpleScalar Tool Set, Version 2.0. Tech. Rep. CS-TR-1997-1342.
- CHANDRAKASAN, A. P., BOWHILL, W., FOX, F., AND BOWHILL, W. July 2000. Design of High-Performance Microprocessor Circuits. In *IEEE Press*.
- CHASE, J. S., LEVY, H. M., LAZOWSKA, E. D., AND BAKER-HARVEY, M. March 1992. Lightweight Shared Objects in a 64-bit Operating System. Tech. Rep. 92-03-09, University of Washington.
- CHEN, J. B., BORG, A., AND JOUPPI, N. P. May 1992. A Simulation-based Study of TLB Performance. In *Proceedings of the 19th International Symposium on Computer Architecture (ISCA '92)*.
- CHENG, R. 1987. Virtual Address Cache in Unix. In *Proceedings of the 1987 Summer Usenix Conference*. 217–224.
- ACM Journal Name, Vol. V, No. N, Month 20YY.

- CHERITON, D. R., SLAVENBERG, G. A., AND BOYLE, P. D. January 1986. Software-Controlled Caches in the VMP Multiprocessor. In *Proceedings of the 13th International Symposium on Computer Architecture (ISCA '86)*.
- COMPAQ. April 1995. Alpha 21164 Microprocessor: Hardware Reference Manual. Digital Semiconductor.
- CORTADELLA, J. AND LLABERIA, J. M. November 1992. Evaluation of A+B=T condition without carry propagation. *IEEE Transactions on Computers*.
- FLAUTNER, K., KIM, N. S., MARTIN, S., BLAAUW, D., AND MUDGE, T. May 2002. Drowsy Caches: Simple Techniques for Reducing Leakage Power. In *In International Symp. on Computer Architecture*.
- GOODMAN, J. AND WOEST, P. June 1988. The Wisconsin Multicube: A New Large-Scale Cache-Coherent Multiprocessor. In *Proceedings of the 15th International Symposium on Computer Architecture (ISCA '88)*.
- GOODMAN, J. R. October 1987. Coherency for Multiprocessor Virtual Address Caches. In *Proceedings of the 2nd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '87)*.
- GOWAN, M. K., BIRO, L. L., AND JACKSON, D. B. 1998. Power Considerations in the Design of the Alpha 21264 Microprocessor. In *Proceedings of the 35th Design Automation Conference (DAC '98)*.
- HENNING, J. L. July 2000. SPEC CPU2000: Measuring CPU Performance in the New Millennium. In *IEEE Computer*. <http://www.specbench.org>.
- HU, Z., JUANG, P., DIODATO, P., KAXIRAS, S., SKADRON, K., MARTONOSI, M., AND CLARK, D. August 2002. Managing Leakage for Transient Data: Decay and Quasi-Static 4T Memory Cells. In *International Symp. on Low-Power Electronics and Design*.
- HUANG, M., RENAU, J., YOO, S.-M., AND TORRELLAS, J. August 2001. L1 Data Cache Decomposition for Energy Efficiency. In *Proceedings of the International Symposium on Low-Power Electronics and Design (ISPLED '01)*.
- INOUE, K., ISHIHARA, T., AND MURAKAMI, K. August 1999. Way-Predicting Set-Associative Cache for High Performance and Low Energy Consumption. In *Proceedings of the International Symposium on Low-Power Electronic Design (ISPLED '99)*.
- IYER, A. AND MARCULESCU, D. March 2001. Power Aware Microarchitecture Resource Scaling. In *Proceedings of the IEEE Design, Automation and Test in Europe (DATE)*.
- JACOB, B. L. AND MUDGE, T. N. February 1997. Software-Managed Address Translation. In *Proceedings of the 3rd International Symposium on High Performance Computer Architecture (HPCA '97)*.
- JACOB, B. L. AND MUDGE, T. N. May 2001. Uniprocessor Virtual Memory without TLBs. In *IEEE Transactions on Computers*. IEEE Press.
- JUAN, T., LANG, T., AND NAVARRO, J. J. August 1997. Reducing TLB power Requirements. In *Proceedings of the International Symposium on Low Power Electronics and Design (ISPLED '97)*.
- KAO, J. T. AND CHANDRAKASAN, A. P. July 2000. Dual-Threshold Voltage Techniques for Low-Power Digital Circuits. In *IEEE JSSC*, 35(7):1009-1018.
- KIN, J., GUPTA, M., AND SMITH, W. M. December 1997. The Filter Cache: An Energy Efficient Memory structure . In *Proceedings of the 30th Annual Symposium on Microarchitecture (MICRO '97)*. IEEE Press.
- KURODA, T. AND SAKURAI, T. August 1996. Threshold-Voltage Control Schemes through Substrate-Bias for Low-Power High-Speed CMOS LSI Design. In *Journal of VLSI Signal Processing Systems*, 30(2/3):191-202.
- KURODA, T., SUZUKI, K., MIRA, S., FUJITA, T., YAMANE, F., SANO, F., AKIHIKO, C., WATANABE, Y., YOSHINORI, M., MATSUDA, K., MAEDA, T., SAKURAI, T., AND TOHRU, F. March 1998. Variable Supply-Voltage Scheme for Low-Power High-Speed CMOS Digital Design. In *IEEE JSSC*, 33(3):454-462.

- LEE, C., POTKONJAK, M., AND MANGIONE-SMITH, W. H. 1997. MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communication Systems. In *Proceedings of the 30th Annual Symposium on Microarchitecture (MICRO '97)*. IEEE Press.
- MA, A., ZHANG, M., AND ASANOVIC, K. July 2001. Way Memoization to Reduce Fetch Energy in Instruction Caches. In *Workshop on Complexity Effective Design, 28th International Symposium on Computer Architecture (ISCA '01)*.
- MONTANARO, J. 1997. A 160-MHz, 32-b, 0.5-W CMOS RISC Microprocessor. In *Digital Technical Journal, vol. 9, Digital Equipment Corporation*.
- MORITZ, C. A., FRANK, M., AND AMARASINGHE, S. 2001. FlexCache: A Framework for Compiler Generated Data Caching. In *Lecture Notes in Computer Science*. Springer Verlag.
- MORITZ, C. A., FRANK, M., LEE, W., AND AMARASINGHE, S. Aug 1999. Hot Pages: Software Caching for Raw Microprocessors. In *MIT-LCS Technical Memo LCS-TM-599*.
- MUTOH, S., DOUSEKI, T., AOKI, Y. M. T., SHINGEMATSU, S., , AND YAMADA, J. August 1995. 1-V Power Supply High-Speed Digital Circuit Technology with Multi-Threshold CMOS Technology. In *IEEE JSSC, 30(8):847-854*.
- PATTERSON, D. A. AND HENNESSY, J. L. 1990. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, San Mateo, CA.
- POWELL, M., YANG, S., FALSAFI, B., ROY, K., AND VIJAYKUMAR, T. 2000. Gated-Vdd: A Circuit Technique to Reduce Leakage in Deep-Submicron Cache Memories. In *Proceedings of ISLPED*.
- POWELL, M. D., AGARWAL, A., VIJAYKUMAR, T. N., FALSAFI, B., AND ROY, K. December 2001. Reducing Set-Associative Cache Energy via Way-Prediction and Selective Direct-Mapping. In *34th Annual Symposium on Microarchitecture (MICRO '01)*. IEEE Press.
- REINMAN, G. AND JOUPPI, N. 1999. An Integrated Cache Timing and Power Model. Compaq WRL Report.
- SAIR, S. AND CHARNEY, M. 2000. Memory Behaviour of the SPEC2000 Benchmark Suite. IBM T. J. Watson Research Center Technical Report.
- SCOTT, M. L., LEBLANC, T. J., AND MARSH, B. D. August 1988. Design Rationale for Psyche, a General-Purpose Multiprocessor Operating System. In *Proceedings of the 1988 International Conference on Parallel Processing*.
- SHIGEMATSU, S. et al. June 1997. A 1-V High-Speed MTCMOS Circuit Scheme for Power-Down Application Circuits. In *IEEE JSSC, 32(6):861-869*.
- SMITH, A. J. September 1982. Cache Memories. In *Computing Surveys, 14(3)*. 473–530.
- UNSAI, O. S., ASHOK, R., KOREN, I., KRISHNA, C. M., AND MORITZ, C. A. December 2001. Cool-Cache for Hot Multimedia. In *34th Annual Symposium on Microarchitecture (MICRO '01)*. IEEE Press.
- UNSAI, O. S., KOREN, I., KRISHNA, C. M., AND MORITZ, C. A. 2002. Cool-Fetch: Compiler-Enabled Power-Aware Fetch Throttling. In *ACM Computer Architecture Letters, Vol 1*.
- VILLA, L., ZHANG, M., AND ASANOVIC, K. 2000. Dynamic zero compression for cache energy reduction. In *International Symposium on Microarchitecture*.
- WANG, W.-H., BAER, J.-L., AND LEVY, H. M. June 1989. Organization and Performance of a Two-Level Virtual-Real Cache Hierarchy. In *Proceedings of the 16th International Symposium on Computer Architecture (ISCA '89)*.
- WHEELER, B. AND BERSHAD, B. N. October 1992. Consistency Management for Virtually Indexed Caches. In *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '92)*.
- WITCHEL, E., LARSEN, S., ANANIAN, C. S., AND ASANOVIC, K. December 2001. Direct Addressed Caches for Reduced Power Consumption. In *34th Annual Symposium on Microarchitecture (MICRO '01)*. IEEE Press.
- WOOD, D. A., EGGERS, S. J., GIBSON, G., HILL, M. D., PENDLETON, J. M., RITCHIE, S. A., TAYLOR, G. S., KATZ, R. H., AND PATTERSON, D. A. January 1986. An In-Cache Address Translation Mechanism. In *Proceedings of the 13th International Symposium on Computer Architecture (ISCA '86)*.

ZHANG, M. AND ASANOVIC, K. December 2000. Highly-Associative Caches for Low-Power Processors. In *Kool Chips Workshop, 33rd Annual Symposium on Microarchitecture (MICRO '00)*.

**Algorithm 1** Hotlines Algorithm: Conservative and Optimistic

---

```

/* For each routine, start with the first basic block */
for each routine do
  E = entry basic block;
  for all hotline registers x: 1 to h do
    hl_access[x] = NULL;
  end for
  Hotline Annotate block E;
end for

/* procedure to Hotline Annotate a block X */
/**** The Conservative Hotlines Algorithm would skip through ****/
/**** pointer-based accesses and non-affine array accesses: ****/
for each access A in X do
  distance = ∞;
  /* find the closest previous access: */
  for x from 1 to h do
    if (proximity(A, hl_access[x]) < distance) then
      closest_reg = x and update distance;
    end if
  end for
  /* if accesses close enough, assign the same register */
  if distance ≤ CacheLineSize/2 then
    map A through hotline register closest_reg;
    hl_access[closest_reg] = A;
    update the hotline register LRU list;
  else
    /* otherwise map through LRU hotline register */
    map A through hotline register LRU_reg;
    hl_access[LRU_reg] = A;
    update the hotline register LRU list;
  end if
end for
workList = successors of X;
while !empty(workList) do
  B = next basic block in workList;
  if B.annotated then
    continue;
  end if
  /* Traverse through the CFG by making recursive calls */
  Hotline Annotate B;
  B.annotated = true;
end while

/* this procedure finds the proximity between two accesses x and y */
proximity(access x, y) {
if x and y are same array accesses with their indices constant c apart then
  return c * element-size;
end if
if x and y are fields of the same structure variable then
  return difference between field offsets;
end if
if x and y are scalar variables declared distance d apart in the same symbol-table then
  return d;
end if
return ∞;
}

```

---