Cool-Mem: Combining Statically Speculative Memory Accessing with Selective Address Translation for Energy Efficiency

Raksit Ashok Saurabh Chheda Csaba Andras Moritz

Department of Electrical and Computer Engineering
University of Massachusetts, Amherst, MA 01003

{rashok, schheda, andras}@ecs.umass.edu

ABSTRACT

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. It combines statically speculative cache access modes, a dynamic CAM based Tag-Cache used as backup for statically mispredicted accesses, various 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, the approach removes the burden of provable correctness in compiler analysis passes that extract static information. This makes Cool-Mem applicable for large and complex applications, without having any limitations due to complexity issues in the compiler passes or the presence of precompiled static libraries. Based on extensive evaluation, for both SPEC2000 and Mediabench applications, 12% to 20% total energy savings are obtained in the processor, with performance ranging from 1.2% degradation to 8% improvement, for the applications studied.

1. INTRODUCTION

The memory system is a major source of power consumption in contemporary processors. For example, the caches and translation look-ahead buffers (TLB) combined consume 23% of the total power in the Alpha 21264 [9], and caches draw 42% of the energy in the StrongARM 110 [19]. With the current trend of ever-increasing on-chip cache sizes, the fraction of the power consumed by caches is likely to further increase. 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.

The architectural remedies proposed (e.g., in the context of caches [16, 10, 11]) 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 new smart ways to reduce power consumption and in many cases even improve application performance.

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, each cache access requires a TAG check, 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 virtually tagged and indexed 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.

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 passes is removed. 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 that it provides 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 [27] and Mediabench [17] applications, we obtain energy savings from 11% to 20% in the processor. The performance obtained ranges from 1.2% degradation to 8% improvement.

The rest of this paper is structured as follows. Section 2 presents the related work as well as 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, and we conclude in Section 8.

2. PREVIOUS WORK

Previous research focusing on TLB power consumption includes the work by Juan et al. [15], where they compare 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. [31] propose 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. Recent papers by Jacob and Mudge [14, 13] also propose a virtually addressed caching architecture with software based cache miss handler, but evaluate only from a performance perspective.

A sizable amount of work has been done toward improving the energy-efficiency of caches. Kin et al. [16] propose 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 by Inoue et al. [11] saves energy on correct prediction by accessing only the matching way instead of all the ways in a set-associative cache. A recent paper by Huang et al. [10] also uses a similar way-prediction scheme. Their cache partitioning scheme includes a specialized stack cache and compiler implementation concerns are addressed. Powell et al. [23] combine way prediction with selective direct mapping to reduce cache energy consumption. Ma et al. [18] propose a deterministic way-memoization scheme as an alternative to way-prediction. Zhang and Asanovic [32] examine Content Addressable TAG (CAM) caches for low power. These techniques are purely architecture based, in contrast to our combined compiler-architecture approach.

A compiler enabled scheme, has been proposed by Moritz et al. [20, 21] in the context of software caches for MIT-RAW processors. Our previous work for embedded systems [28] 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 handle complex program structures, support different levels of speculation, and focuses on multi-level memory systems incorporating virtual memory support.

Compiler enabled techniques targeting cache energy also include the recent work by Witchel et al. [30]. Their scheme saves the tag-check energy when the compiler can *guarantee* an access will be to the same line as an earlier access. The work is similar to the predictable cache access mechanism described for Raw processors in [21]. Additionally, their technique works primarily for applications with predictable data accesses, focusing mainly on affine array accesses and simple loop structures, and affects code size.

Physically addressed caches are becoming increasingly unfeasible with growing cache sizes [22]. 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 Cool-Mem design).

Virtually addressed caches come with their share of problems. The well known *synonym problem* is detailed in the work by Goodman [8]. Both hardware and software solutions to these problems have already been proposed [8, 29]. This problem is a primary concern in a virtual-virtual cache hierarchy, but could be easily dealt in a hierarchy involving a virtually-addressed first level cache and a physically addressed second level cache [29].

Additionally, the problem can also be overcome by disallowing aliasing altogether. This can be accomplished by providing a global address space model. Another way is to force shared data to align in the cache, or require that shared data be non-cacheable [5]. Furthermore, other solutions allow arbitrary aliases but implement a consistency protocol in hardware [29]. Other hardware based techniques to deal with the synonym problem are based on ideas such as back pointers [29] or dual tag sets (having both a physical and a virtual tag), or reverse translation tables [26] that translate physical addresses into virtual addresses.

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 example, 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 [4]. Single address space operating systems like Opal [3], that use global addresses, would not have to deal with aliasing. 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. DMA accesses can be supported by flushing affected cache blocks before the transfer

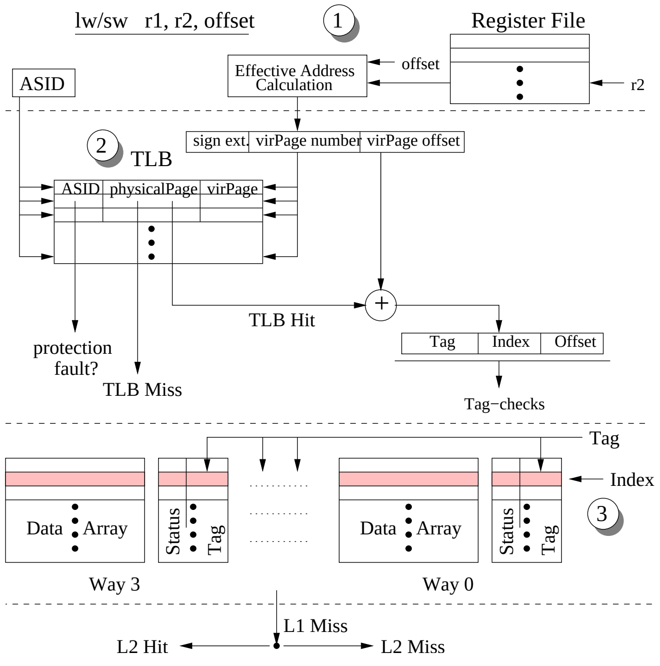


Figure 1: The baseline memory system. All accesses require a TLB access and tag-checks.

or having regions in the cache that are non-cacheable.

To the best of our knowledge, Cool-Mem 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.

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 [6] 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 64-bit virtual address is then 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. 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

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 two different 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. As second level, we evaluate both a physically-indexed and a virtually-indexed cache. As described in Section 2, 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. 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 towards lower levels in the cache hierachy is shown in Figure 3. As address translation consumes a significant fraction of the energy consumed in the memory system, 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. 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.

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 directly address the cache, and the conventional associative lookup mechanism is used only on Tag-Cache misses. Integration of

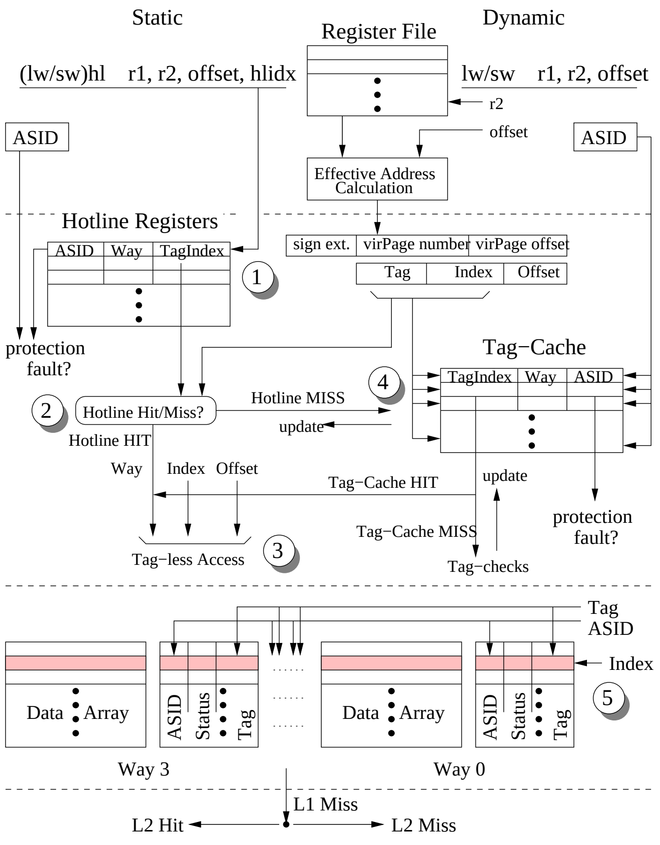


Figure 2: The Cool-Mem memory system.

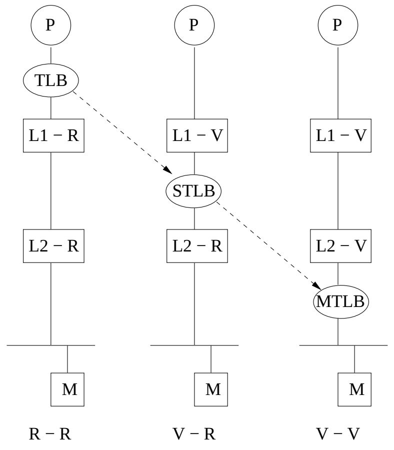


Figure 3: Various cache organizations, with address translation moved toward lower levels in the memory hierarchy.

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 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 [7] the condition that the hotline path evaluates can be done very efficiently without carry propagation. 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 [24], 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 [12], 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 re-balance the delay differences between the tag and data-array paths [24]. 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. 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.

4.2 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

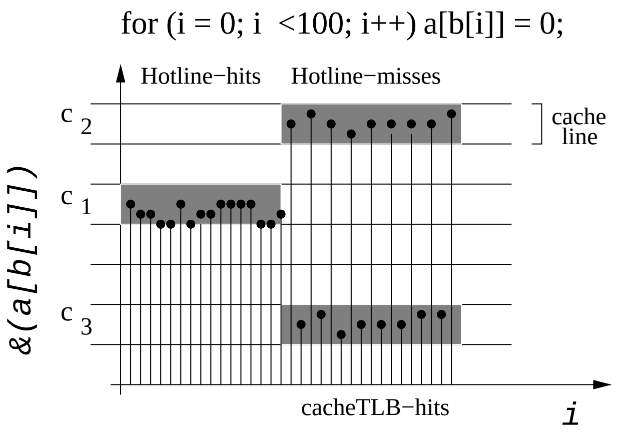


Figure 4: Hotline-misses hitting the Tag-Cache. We found that aggressively speculative hotline accesses that cause hotline-misses, hit the Tag-Cache with good probability.

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. For the pattern shown in Figure 4 all the alternating accesses will be hotline hits but are captured by the Tag-Cache due to its associative nature. In general, the Tag-Cache 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. As seen in Figure 2, each Tag-Cache entry is exactly the same as a hotline register, and performs the same functions, but dynamically.

4.3 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.

5. COOL-MEM COMPILER

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 [21, 20, 28], 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.

We 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 hotline pass works on a per procedure basis, see Algorithm 1. The input to this algorithm is the set of hotline registers, $\{r_1, r_2, ..., r_h\}$, 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. We now illustrate the working of this algorithm with some simple examples.

```
(a)  for (i = 0; i < 100; i++)
        a[i]{1} = a[i+1]{1} + a[i+100]{2} + a[i+103]{2}

(b)  for (i = 0; i < 100; i++)
        a[i]{..} = a[i]{..}*var1.field1{1} + a[i+1]{..}*var1.field2{1}
                   + a[i+2]{..}*var1.field10{2}
```

Figure 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.

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 line is 64 bytes, implying a threshold distance of 32 bytes. The hotline analysis assigns a[i] hotline register r_1. Arriving at 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. For a[i+100] however, the closest access is beyond the threshold (a[i+1] is 396 bytes from a[i+100]), and hence a[i+100] is mapped through a different hotline register 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

Algorithm 1 Conservative and Optimistic Hotlines

```
/* 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 Algorithm skips through ****/
/**** pointer-based 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

/* procedure to find the proximity between 2 accesses */
/* 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 ∞;
```

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. 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. An even more conservative approach may not hotline even affine array accesses, if the stride information is unknown at compile-time.

```
/* Irregular array accesses */
for (i = 0; i < 100; i++)
   a[b[i]]{not mapped} = 0;
```

(a)

```
/* Pointer−based accesses */
for (i = 0; i < 100; i++)    {
   p−>val{x} = a[i]{..};
   p{x} = p−>next{x};
}
```

(b)

Figure 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.

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.

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

6. EXPERIMENTAL FRAMEWORK

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 [2] simulator with all the required modifications in place.

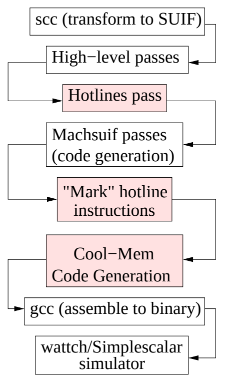


Figure 7: Shaded steps are introduced by Cool-Mem.

We have used the SimpleScalar [2] simulator with Wattch [1] 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.

For evaluation, we use the CPU2000 [27] and Mediabench [17] applications. Six Mediabench and six CPU2000 benchmarks have been randomly chosen, see Table 1. 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 [25].

Table 1: Benchmarks used for evaluation

Benchmark	Description
ADPCM	Audio compression (Mediabench)
EPIC	Image compression (Mediabench)
G721	Voice compression (Mediabench)
JPEG	Image decompression (Mediabench)
MPEG	Video decompression (Mediabench)
RASTA	Speech recognition (Mediabench)
MCF	Combinatorial optimization (CPUInt00)
PARSER	Synthetic English parser (CPUInt00)
VPR	FPGA circuit place and route (CPUInt00)
AMMP	Computational chemistry (CPUFP00)
ART	Image recognition/neural nets (CPUFP00)
EQUAKE	Seismic wave simulation (CPUFP00)

Table 2: Baseline System Parameters

Parameter	Baseline value
Processor Speed	1.5Ghz
Process technology	0.18μm, 2V
ITLB	64-entry Fully-assoc.
DTLB	64-entry Fully-assoc.
TLB-miss penalty	20 cycles
L1 D-cache	64k, 4-way, 64byte line
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 3: 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 2. First we show Cool-Mem results with identical system configuration as the baseline (see Table 2), for both the virtual-real Cool-Mem architecture (v-r) and the virtual-virtual Cool-Mem architecture (v-v). Next, we present sensitivity results by changing certain baseline parameters.

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 8(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 8(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 8(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 8(c) shows the L1 D-Cache access patterns for CPU2000 and MediaBench applications. The lower bars are the 1-cycle static hits, the middle bars are 2-cycle TAG-Cache hits, and the upper bars are TAG-Cache misses that are L1 hits, and the rest are L1 misses. Figure 8(d) shows the relative energy consumption in the L1 D-Cache, broken down into various components. 100% corresponds to the 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 next present overall energy and performance results (these results are different for v-r and v-v). In Figure 9(b), we show the performance gains in v-r and v-v. For the v-r design, we get performance ranging from 1.2% degradation to 8% improvement. On average, a miniscule 0.36% performance degradation for CPU2000 and 3.6% improvement for MediaBench is achieved. Given the good static hit rates, one would have expected performance improvement for all applications. The 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. When L1 miss rate is high enough, the performance penalty on L2 access can outweigh the benefits of Hotline hits.

The v-v design has higher performance benefits: 1.9% for CPU2000 and 3.7% for MediaBench applications on average,

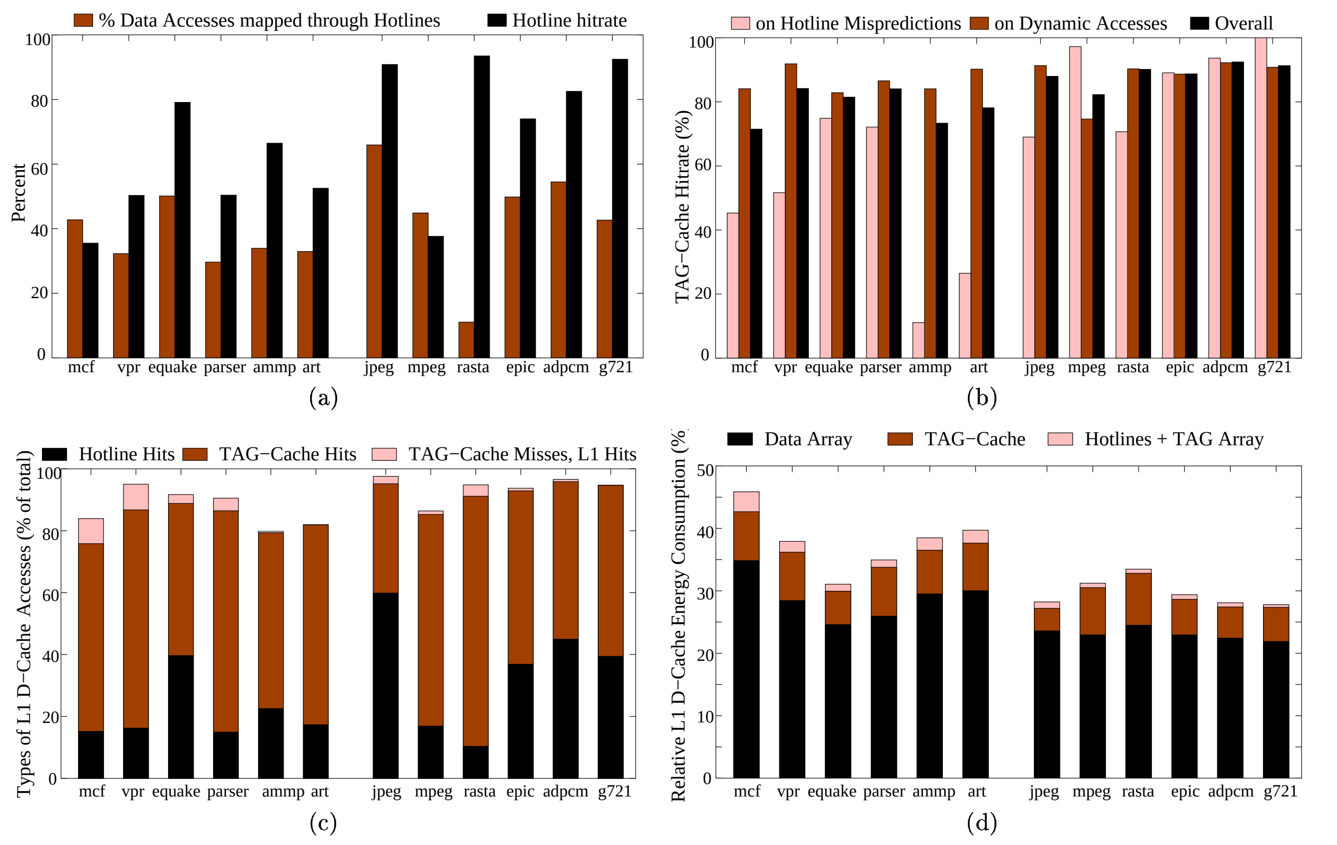


Figure 8: Baseline results

with maximum gain at 8%. 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 9(a) shows the total energy savings, and how much of this saving comes from the caches, the TLBs, and the clock. Clock energy is directly proportional to chip-activity and therefore as we reduce the activity by reducing cache and TLB energy consumptions, clock dissipation also goes down. Excellent energy savings are obtained, averaging 18% for CPU2000 and 16% for MediaBench, in both v-r and v-v designs.

The Energy-delay product is plotted in Figure 9(c). Due to better performance gain in v-v compared to v-r, v-v has a slight advantage in the energy-delay product: The product is 80% for CPU2000 in v-v compared to 82% in v-r, and 80% for MediaBench in both v-v and v-r designs.

7.2 Sensitivity Analysis

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

7.2.1 Optimistic Vs Conservative Hotlines

Figure 10 shows the Hotline hit rates for the two hotline algorithms. 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", "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.

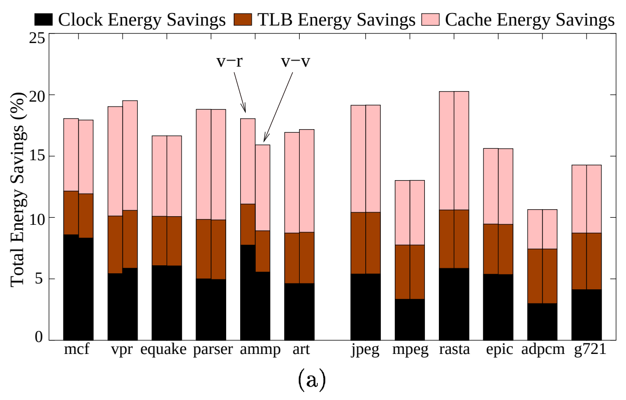
7.2.2 Hotline Register-file and TAG-Cache sizes

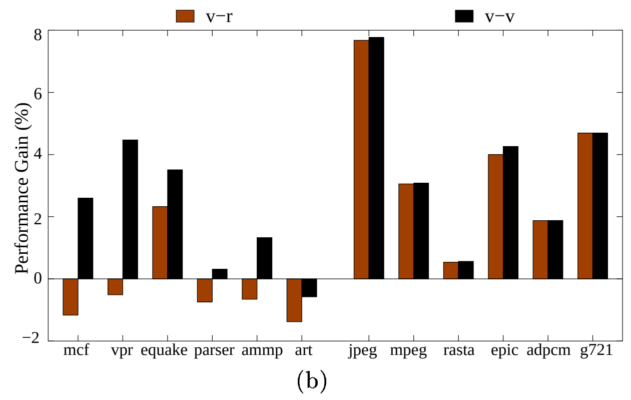
Figure 11(a) looks at static hit rate with increasing number of hotline registers. As expected, we see an improvement in the hit rate with increasing number of hotline registers. Saturation occurs around 16 registers, with minor improvement going to 32 registers.

Figure 11(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.

7.2.3 Cache Line Size

Figure 12 shows the sensitivity of our results to increasing L1 D-Cache line sizes. The static hit rate is seen to be





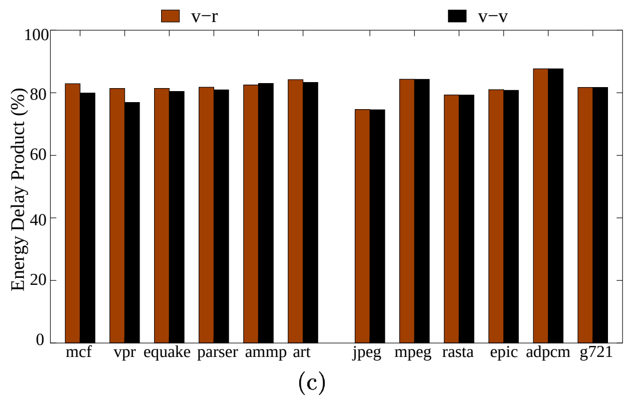


Figure 9: Baseline Results

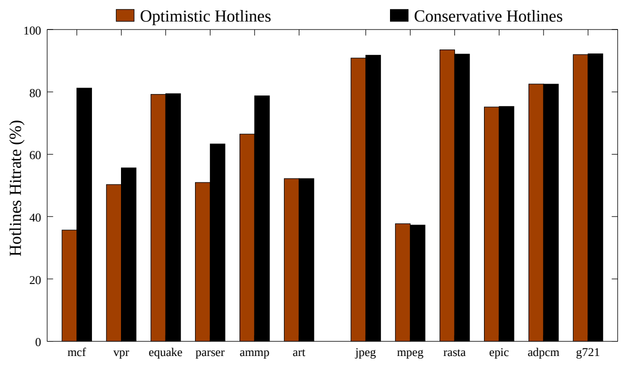
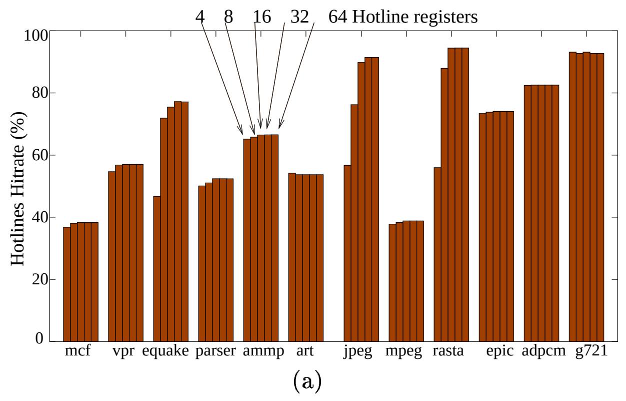


Figure 10: Varying levels of Speculation in the CoolMem Compiler



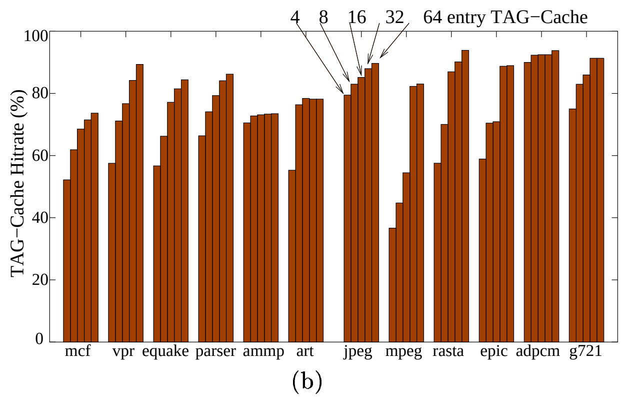


Figure 11: Sensitivity to Hotline register-file and TAG-Cache sizes

fairly sensitive to cache line size: As line size increases, the Hotline hit-rate is also seen to increase. The Tag-Cache hit rate also increases with increasing cache line size, as shown in Figure 12. This is expected, because the likelihood of an access mapping to a particular cache line increases as the line size increases.

Figure 12 also shows the 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.

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. We observe from the energy delay graph in Figure 12 that a line size of 64bytes generally gives the best result.

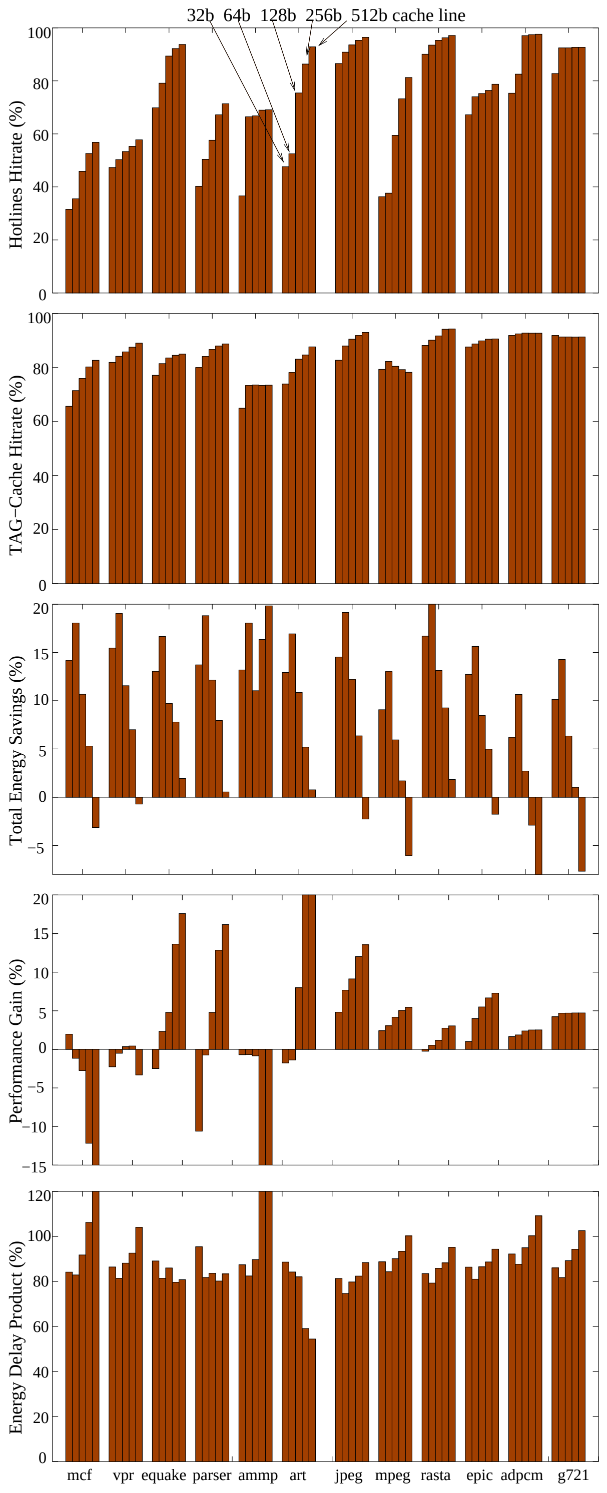


Figure 12: Sensitivity to Cache Line Size

7.2.4 *Cache Size*

Figure 13 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

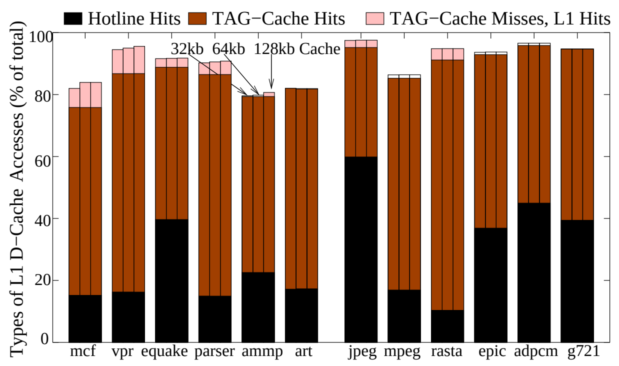


Figure 13: Hitrate sensitivity to Cache size

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.

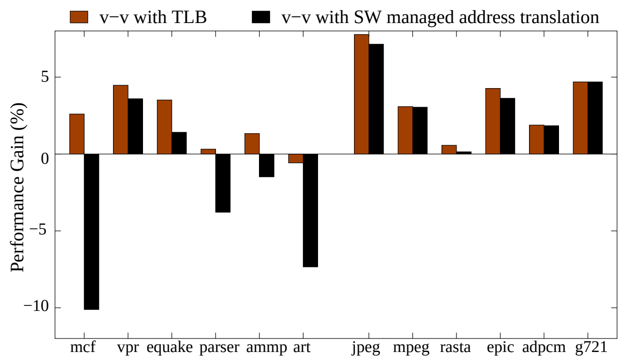


Figure 14: v-v performance: with TLBs vs with software managed address translation

7.2.5 *TLB vs Software Managed Address Translation*

Figure 14 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 associated with virtual addressing can be effectively taken care of here. Even with software-based address translation, performance is gained on all MediaBench applications and half of the CPU2000 applications.

8. 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. Cool-Mem achieves significant energy reduction in the processor, ranging from 11% to 20%, with performance ranging from 1.2% degradation to 8% 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.

9. REFERENCES

[1] D. Brooks, V. Tiwari, and M. Martonosi. Wattch: a framework for architectural-level power analysis and optimizations. In *Proceedings of the 27th International Symposium on Computer Architecture (ISCA '00)*, June 2000.

[2] D. C. Burger and T. M. Austin. The SimpleScalar Tool Set, Version 2.0. Technical Report CS-TR-1997-1342, 1997.

[3] J. S. Chase, H. M. Levy, E. D. Lazowska, and M. Baker-Harvey. Lightweight Shared Objects in a 64-bit Operating System. Technical Report 92-03-09, University of Washington, March 1992.

[4] J. B. Chen, A. Borg, and N. P. Jouppi. A Simulation-based Study of TLB Performance. In *Proceedings of the 19th International Symposium on Computer Architecture (ISCA '92)*, May 1992.

[5] R. Cheng. Virtual Address Cache in Unix. In *Proceedings of the 1987 Summer Usenix Conference*, pages 217–224, 1987.

[6] C. Corporation. Alpha 21164 Microprocessor: Hardware Reference Manual. Digital Semiconductor, April 1995.

[7] J. Cortadella and J. M. Llaberia. Evaluation of A+B=T condition without carry propogation. *IEEE Transactions on Computers*, November 1992.

[8] J. R. Goodman. Coherency for Multiprocessor Virtual Address Caches. In *Proceedings of the 2nd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '87)*, October 1987.

[9] M. K. Gowan, L. L. Biro, and D. B. Jackson. Power Considerations in the Design of the Alpha 21264 Microprocessor. In *Proceedings of the 35th Design Automation Conference (DAC '98)*, 1998.

[10] M. Huang, J. Renau, S.-M. Yoo, and J. Torrellas. L1 Data Cache Decomposition for Energy Efficiency. In *Proceedings of the International Symposium on Low-Power Electronics and Design (ISPLED '01)*, August 2001.

[11] K. Inoue, T. Ishihara, and K. Murakami. 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)*, August 1999.

[12] A. Iyer and D. Marculescu. Power Aware Microarchitecture Resource Scaling. In *Proceedings of the IEEE Design, Automation and Test in Europe (DATE)*, March 2001.

[13] B. L. Jacob and T. N. Mudge. Software-Managed Address Translation. In *Proceedings of the 3rd International Symposium on High Performance Computer Architecture (HPCA '97)*, February 1997.

[14] B. L. Jacob and T. N. Mudge. Uniprocessor Virtual Memory without TLBs. In *IEEE Transactions on Computers.* IEEE Press, May 2001.

[15] T. Juan, T. Lang, and J. J. Navarro. Reducing TLB power Requirements. In *Proceedings of the International Symposium on Low Power Electronics and Design (ISPLED '97)*, August 1997.

[16] J. Kin, M. Gupta, and W. M. Smith. The Filter Cache: An Energy Efficient Memory structure . In *Proceedings of the 30th Annual Symposium on Microarchitecture (MICRO '97).* IEEE Press, December 1997.

[17] C. Lee, M. Potkonjak, and W. H. Mangione-Smith. MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communication Systems. In *Proceedings of the 30th Annual Symposium on Microarchitecture (MICRO '97).* IEEE Press, 1997.

[18] A. Ma, M. Zhang, and K. Asanovic. Way Memoization to Reduce Fetch Energy in Instruction Caches. In *Workshop on Complexity Effective Design, 28th International Symposium on Computer Architecture (ISCA '01)*, July 2001.

[19] J. Montanaro. A 160-MHz, 32-b, 0.5-W CMOS RISC Microprocessor. In *Digital Technical Journal, vol. 9, Digital Equipment Corporation*, 1997.

[20] C. A. Moritz, M. Frank, and S. Amarasinghe. FlexCache: A Framework for Compiler Generated Data Caching. In *Lecture Notes in Computer Science.* Springer Verlag, 2001.

[21] C. A. Moritz, M. Frank, W. Lee, and S. Amarasinghe. Hot Pages: Software Caching for Raw Microprocessors. In *MIT-LCS Technical Memo LCS-TM-599*, Aug 1999.

[22] D. A. Patterson and J. L. Hennessy. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann, San Mateo, CA, 1990.

[23] M. D. Powell, A. Agarwal, T. N. Vijaykumar, B. Falsafi, and K. Roy. Reducing Set-Associative Cache Energy via Way-Prediction and Selective Direct-Mapping. In *34th Annual Symposium on Microarchitecture (MICRO '01).* IEEE Press, December 2001 (To Appear).

[24] G. Reinman and N. Jouppi. An Integrated Cache Timing and Power Model. Compaq WRL Report, 1999.

[25] S. Sair and M. Charney. Memory Behaviour of the SPEC2000 Benchmark Suite. IBM T. J. Watson Research Center Technical Report, 2000.

[26] A. J. Smith. Cache Memories. In *Computing Surveys, 14(3)*, pages 473–530, September 1982.

[27] The standard performance evaluation corporation. In *http://www.spec.org*, 2000.

[28] O. S. Unsal, R. Ashok, I. Koren, C. M. Krishna, and C. A. Moritz. Cool-Cache for Hot Multimedia. In *34th Annual Symposium on Microarchitecture (MICRO '01).* IEEE Press, December 2001 (To Appear).

[29] W.-H. Wang, J.-L. Baer, and H. M. Levy. Organization and Performance of a Two-Level Virtual-Real Cache Hierarchy. In *Proceedings of the 16th International Symposium on Computer Architecture (ISCA '89)*, June 1989.

[30] E. Witchel, S. Larsen, C. S. Ananian, and K. Asanovic. Direct Addressed Caches for Reduced Power Consumption. In *34th Annual Symposium on Microarchitecture (MICRO '01).* IEEE Press, December 2001 (To Appear).

[31] D. A. Wood, S. J. Eggers, G. Gibson, M. D. Hill, J. M. Pendleton, S. A. Ritchie, G. S. Taylor, R. H. Katz, and D. A. Patterson. An In-Cache Address Translation Mechanism. In *Proceedings of the 13th International Symposium on Computer Architecture (ISCA '86)*, January 1986.

[32] M. Zhang and K. Asanovic. Highly-Associative Caches for Low-Power Processors. In *Kool Chips Workshop, 33rd Annual Symposium on Microarchitecture (MICRO '00)*, December 2000.