34th International Symposium on Microarchitecture (MICRO-34), Austin, TX, December 2001.

Direct Addressed Caches for Reduced Power Consumption

Emmett Witchel, Sam Larsen, C. Scott Ananian and Krste Asanović

MIT Laboratory for Computer Science
{witchel, slarsen, cananian, krste}@lcs.mit.edu

Abstract. A *direct addressed cache* is a hardware-software design for an energy-efficient microprocessor data cache. Direct addressing allows software to access cache data without a hardware cache tag check. These tag-unchecked loads and stores save the energy of a tag check when the compiler can guarantee an access will be to the same line as an earlier access. We have added support for tag-unchecked loads and stores to C and Java compilers. For Mediabench C programs, the compiler eliminates 16–76% of data cache tag accesses, with half of the benchmarks avoiding over 40% of the data tag checks. For SPECjvm98 Java programs, the compiler eliminates 18–63% of data cache tag checks. These tag check reductions translate into data cache energy savings of 9–40%, and overall processor and cache energy savings of 2–8%.

1 Introduction

Reducing energy consumption is an important goal for processors that will be used in battery-powered devices. Caches consume a large portion of total energy in processors targeted at low-power applications. For example, 16% of the total processor and cache power for the StrongARM microprocessor is dissipated in the data cache [6].

Commercial low-power processors usually employ associative caches [2, 6, 10, 13, 18]. For associative caches, a significant portion of the total access energy is spent checking multiple tags to find where data resides in the cache. For example, the highly-associative low-power cache designs used by the StrongARM and Xscale processors expend over 50% of the total cache access energy in the tag check [20].

In this paper, we propose a new hardware-software interface to reduce the energy cost of accessing cache data. *Direct addressing* allows

software to access cache data without the hardware performing a cache tag check. These *tag-unchecked loads and stores* save the energy of performing a tag check when the compiler can guarantee an access will be to the same line as an earlier access. If the compiler cannot determine this information, or if cache lines are evicted due to interrupts or cache invalidations, direct addressing gracefully degrades back to conventional tag-checked accesses.

We have implemented compiler support for direct addressing in the SUIF C compiler [8], and in FLEX, a Java bytecode-to-native compiler [7]. We evaluate our compiler algorithms using C programs from Mediabench, and Java programs from SPECjvm98. Our results show we can eliminate 16–76% of all data cache tag accesses in C, and 18–63% of data cache tags checks in Java. We have developed a detailed energy model of a power-optimized microprocessor and caches. The reduction in cache tag checks results in data cache energy savings of 9–40% in C and 9–31% in Java. The total processor plus cache energy savings is 2–8%.

The paper is structured as follows. First we review current cache design in Section 2. Section 3 describes the changes needed to implement direct addressing. General compiler algorithms to support direct addressing are discussed in Section 4. The algorithms and results specific to C are described in Section 5, and the algorithms and results for Java in Section 6. Section 7 compares direct addressing to hardware schemes that remove tag checks. Finally, we discuss related work and conclude.

2 Low-power cache designs

Figure 1 shows the structure of a conventional virtually-indexed, virtually-tagged set-associative RAM-tagged cache (for brevity, only virtual caches are considered here, but direct addressing can be applied to other types of caches). An index taken from the virtual address is used to select a set consisting of several ways, and the tag field of the virtual address is compared against the tags in all ways to determine the location of the data. An n-way associative cache performs n tag checks and n data reads in parallel, discarding all but one of the data values depending on the tag compares.

An alternative approach, used in many low-power microprocessors [2, 6, 10, 13, 18], is to store the tags in content-addressable memory (CAM). The tag is broadcast across the cache lines and only the line whose tag matches has its data read out. The energy consumption of a 32-way CAM-tag search is approximately the same as a 2-way set

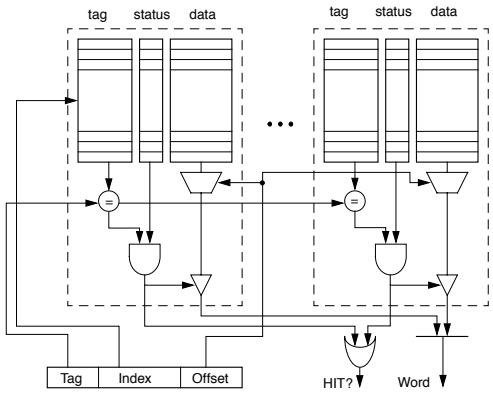

Fig. 1. A set-associative RAM-tag cache.

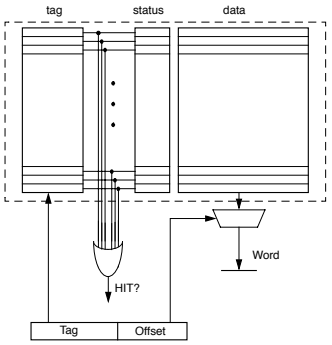

Fig. 2. A highly-associative CAM-tag cache subbank.

associative RAM-tag search [20, 2] but has lower miss rates. Caches are often subbanked to save energy and reduce delay, and a CAM-tag cache subbank is shown in Figure 2. Although CAM-tag caches reduce miss rates and hence total absolute memory access energy, they expend relatively greater energy in tag checks. Detailed HSpice simulations of a 16 KB CAM-tagged data cache divided into 1 KB subbanks, shows that the tag check consumes 54% of cache energy for loads and 43% for stores.

For both RAM and CAM tag caches, searching tags is expensive. If we could shortcut the process, by letting software tell the hardware in which way the line is located, we could save significant energy. The problem is how to let software directly access cache lines without com-

promising inter-process protection and while preserving correct operation in the face of cache replacements or other cache coherence actions.

3 Direct addressing

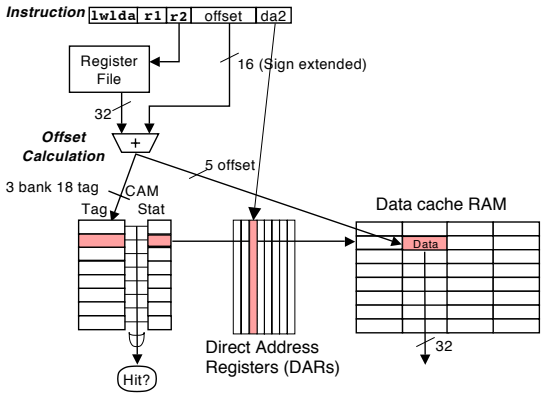

Fig. 3. A CAM-tagged data cache with direct addressing. The `lwlda` instruction causes `da2` to memoize the location of the data. A subsequent `lwda` that used `da2` would not power up the CAM bank on the left, but use the shaded DAR to pick this line.

Our approach to eliminating tag checks is to let software tell the hardware to remember the location of a cache line, so when software accesses the line again, hardware can access the data directly without searching tags. We augment the processor state with some number of *direct address registers* (DARs). These registers are set and used by software, and contain enough information to specify the exact location of a cache line in the cache data RAM as well as a valid bit. The exact width and data layout of the DARs is hidden from software to avoid exposing the implementation-dependent structure of the cache. In particular, software is only made aware of the length of a cache line, but not the total cache capacity or associativity.

Table 1 shows the instruction extensions for using DARs. Software places values in the DARs as an optional side-effect of performing a load or store. A tag-unchecked load or store specifies a full effective virtual address in addition to a DAR number. If the DAR is valid, its contents are used to avoid a tag search; if it is invalid, hardware

Instruction	*Explanation*
`(l|s)wlda rt, off(rs), da`	Load or store word, load direct address. Perform regular load or store, and also set the direct address register `da` to the location of the referenced line.
`(l|s)wda rt, off(rs), da`	Load or store word, using direct address. Data from the cache line pointed to by `da` is transferred to register `rt` (or the contents of `rt` is stored into the line specified by `da`). The line offset bits of `rs + off` are used to pick the proper word in the line. If `da` is invalid, the instruction acts like `(l|s)wlda`, accessing memory and setting the `da` register.
`jr.dainv rs, da_mask`	Jump register and invalidate direct address registers. It acts like a jump register instruction, and also clears the valid bit on the DARs specified in the bitmask. It is used on function return to invalidate the DARs used by the function.

Table 1. A table of instruction set extensions for manipulating direct address registers. MIPS is the base ISA and a machine with 8 DARs is described. Only word accesses are shown, but half-word and byte accesses are handled analogously.

falls back to a full tag search using the entire virtual address. The implementation described here uses a separate DAR specifier in each instruction, which takes 3 bits from the 16-bit immediate offset. An alternative encoding is to implicitly associate a DAR with some set of base registers, which reduces ISA changes at the cost of complicating compiler register allocation. We do not consider this option further in this paper.

Direct addressing is only used for data caches. Instruction caches have very regular access patterns and are only accessed via the program counter, and hence are amenable to software-invisible micro-architectural techniques to remove tag checks [16, 18, 19].

As an example, consider the function entry code in Figure 4, and a transformation of that code using direct addressing. The `swlda` instruction sets up the `da0` DAR, which is then used by the following `swda`

instructions to eliminate cache tag checks. Note that no additional instructions were added and that performance is identical.

```
Old Code            New Code
sub $sp,64          sub   $sp,64
sw  $ra,60($sp)     swlda $ra,60($sp),$da0
sw  $fp,56($sp)     swda  $fp,56($sp),$da0
sw  $s0,52($sp)     swda  $s0,52($sp),$da0
```

Fig. 4. Example function entry code transformed to use DARs.

3.1 DAR implementation

At minimum, a DAR need only record the matching way within the cache set. In this case, the effective address is used to obtain the subbank number, the set index, and the offset within the cache line. In some implementations, however, it will be advantageous to also record subbank and set index information in the DARs and to physically distribute the DARs among the cache subbanks. This avoids recalculating and retransmitting these portions of the virtual address for tag-unchecked accesses.

The DARs incur additional area, energy, and delay overheads. The primary energy penalty is the parasitic load of the DARs on the signal lines driving the cache, but this should be a negligible fraction of overall cache access energy. The delay penalty is a single mux to select either one of the DARs or the normal cache access signal.

For a RAM-tag cache, the DARs can record way hit/miss information locally in each way (each way is a subbank). For a tag-unchecked access, the DAR specifier is broadcast to the ways, which replay the hit/miss information recorded in the local DAR latches without performing a tag check. The area and energy overhead of the DAR bits is small compared to the cache itself. The delay penalty is only a fraction of a gate delay as the DAR hit/miss signal can be folded into the existing precharged tag comparator.

For a CAM-tag cache, a DAR would be implemented as a unary bit vector with a single bit set on the matching row. Each cache row would locally store one bit per DAR. The DARs would be written with the local hit/miss signals generated by the CAM tag in each row.

For regular accesses, the parasitic energy overhead of the DARs is small because at most only one row's hit signal transitions high and one row's hit signal transitions low on any search. There is an additional energy cost to writing a DAR, where the DAR clock line has to transition high and low, but this overhead is small compared to the saving from not driving multiple bits of address across the tag array when the DAR is next used. As with the RAM-tag cache, the delay penalty is small if the DAR hit/miss signal is folded into the precharged match comparator.

3.2 DAR coherence

The DARs must be kept coherent with the state of the cache. If a line pointed to by a DAR is evicted, the DAR contents are no longer valid and cannot be used in a tag-unchecked access. Lines may be evicted either as a result of cache line replacement, or by external invalidate requests to maintain cache coherence with other processors or DMA I/O traffic.

To maintain coherence, each DAR can be tagged with the address of the cache line to which it points. On any eviction, the DAR tags are searched associatively and matching DARs are invalidated. The next use of an invalid DAR will cause a regular tag-checked access (which will usually miss). The DAR address tags need hold only a portion of the entire address allowing only a partial compare against the victim address, trading off some additional spurious invalidations for reduced complexity. In the extreme case, the DAR tags can be omitted with all DARs invalidated on any eviction.

The validity of the DARs can be checked right after the instruction decode of a tag-unchecked access. If the register is not valid, the access is converted into a regular tag-checked access early in the instruction pipeline, well before reaching the memory access stage. This avoids any additional memory access latency for checking valid bits.

4 Compiler algorithms for using DARs

Direct addressing has been implemented in two compiler systems, a SUIF-based C compiler and the FLEX Java native compiler. This section describes compiler algorithms common to both systems.

Both compilers use the same two step approach to eliminate tag checks with direct addressing. First, find two references, one of which dominates the other, so all paths that cause the subordinate access to be executed cause the dominant reference to be executed first. Second,

prove that the two references always point to the same cache line. The second reference can then skip the tag check, by having the dominant reference write a DAR that the subordinate reference reads. Any other code between the two references, including assignments, control flow, or even function calls, can not affect correctness because hardware will invalidate DARs that point to lines that get evicted between the definition and the use of a DAR (as discussed in Section 3.2 above).

Both compilers control the stack pointer, ensuring it remains aligned to a cache boundary to simplify the determination of when two stack variables are on the same cache line. This allows easy transformation of function entry/exit code (as in Figure 4), spill code, parameter passing code, and access to automatic variables. The C and Java compilers use different methods to determine if two references to non-stack data (heap and static data) are to the same cache line. These are discussed in Sections 5.1 and 6.1 respectively.

4.1 DAR allocation

Each dominant reference with at least one subordinate reference to the same cache line is marked as a candidate for a DAR. The DAR allocation problem is an instance of the standard register allocation problem — DAR candidates that are live at the the same program point interfere and need to be allocated to different hardware DARs. DAR allocation is simpler than processor register allocation because DARs can not be spilled. Instead of spilling, a DAR is simply not allocated to a problematic DAR candidate.

The metric of utility we use for allocation is the number of tag checks eliminated by a certain DAR candidate minus the number of tag checks eliminated by the DAR candidates with which it interferes. This causes small, non-interfering ranges to get good coverage, and the most important variables in regions of heavy DAR use are prioritized.

4.2 DARs and calling conventions

The compilers analyze one function at a time, and the DARs are caller invalidated—at function exit, the compiler invalidates the DARs used in the function. If a function has a DAR live (say `da3`), and it makes a function call, the called function might invalidate `da3`, forcing a tag check on the use of that register. To reduce the impact of inter-procedural DAR invalidates, we randomly permute the DAR numbers used by the allocator. So one function might use registers in the order 7,2,3,6,0,5,1,4, another in the order 5,1,0,7,2,6,4,3.

Random permutation is much simpler than inter-procedural analysis, and makes collisions between register numbers much less likely than if every function used the same order. Interference is very low, and is quantified for C programs in Table 2 and for Java programs in Table 3.

5 C compiler implementation

We employ alignment and distance analysis for C to determine if two references are to the same cache line. This section first describes alignment and distance analysis in our C compiler, and then discusses the results of our experiments.

5.1 Alignment analysis in C programs

Alignment analysis attempts to determine the address alignment of each static memory reference relative to a cache line boundary. A value of 24 would indicate that the associated memory reference always accesses an address that is 24 bytes offset from the start of the cache line. A load or store instruction is considered *aligned* when its cache alignment is the same for each dynamic execution of the instruction. For instance, a global scalar resides in a static memory location and therefore always occupies a set alignment within the cache. For the majority of memory operations however, this will not be the case. Consider the loop in Figure 5(a). Here, the store instruction will access sequential cache locations in each loop iteration and is therefore *unaligned*.

In order to increase the percentage of aligned memory operations, our compiler performs a series of alignment-increasing transformations. One of the most important is loop unrolling. The code in Figure 5(b) shows the original loop with unrolling. After unrolling the loop by a factor consistent with the size of the cache line, we can guarantee that each memory operation in the loop only accesses the cache with a certain alignment. This is the case in our example assuming that A is an array of 64-bit data, and the cache line size is 32 bytes.

Since inner loops comprise the majority of dynamically executed instructions, it is very important that we uncover as much alignment information as possible from the body of an inner loop. Loop unrolling is effective for array references when the array is a local or global variable. However, if the array in Figure 5 is passed as an argument to the enclosing function, then loop unrolling does not enable the analysis to guarantee alignment for the memory references within the loop since the base of the array is unknown. Even worse is the case when the base

```
for (i=0; i<N; i++) {
  A[i] = 0;
}
```

(a)

```
for(i=0; i<N;
         i += 4) {
A[i + 0] = 0;
A[i + 1] = 0;
A[i + 2] = 0;
A[i + 3] = 0;
}
```

(b)

```
for (i=0; i<N; i++) {
   if (&A[i]
       % line_size == 0)
   break;
    A[i] = 0;
}
for (; i<N; i += 4) {
   A[i + 0] = 0;
   A[i + 1] = 0;
   A[i + 2] = 0;
   A[i + 3] = 0;
}
```

(c)

Fig. 5. (a) A simple loop with a single memory reference. **(b)** After loop unrolling. **(c)** A pre-loop inserted to guarantee alignment in the unrolled loop body.

of the array is actually aligned differently for different invocations of the function.

To overcome this limitation, our compiler inserts a pre-loop that runs for a small number of iterations until the references within the loop reach a known alignment. The code then jumps to an unrolled version of the loop where the alignment of references within the unrolled body are guaranteed (Figure 5(c)). Using this technique, the alignment analysis can determine the alignment for the majority of dynamically executed memory accesses. In order to limit the number of pre-loop iterations that are executed, our compiler also uses profile-driven feedback to determine the best conditions to begin execution of the unrolled loop.

One disadvantage of using loop unrolling to obtain alignment information is that too much unrolling can increase I-cache pressure [11]. We did not measure the impact of this effect.

5.2 Distance analysis

Distance analysis attempts to determine the byte distance between the addresses of two static memory references. The algorithm is implemented as a dataflow analysis that operates on low-level address calculations. If the difference between address calculations is a constant, then we know the distance between the references.

In the initial compiler passes, when array accesses are represented at a high level, we tag them with their source array to aid in distance analysis. We use this tag once the array access has been decomposed into pointer manipulation. For accesses of the form $A[i]$ and $A[i + c]$, our tagging allows us to compute the distance as c. This pattern occurs very frequently in unrolled loops.

We deal with aliasing using local information. To be conservative, we assume a pointer variable can point to any globally visible address. So a DAR definition and use will not span a pointer store to a base with a globally visible address.

Once we know the distance, we can use the alignment to determine if two references are to the same cache line. We find the alignment of the dominant reference relative to the cache line boundary and then find the distance between the subordinate access and the dominant access. Simple arithmetic indicates if the references are on the same cache line. An important special case is when the distance is 0, in which case we do not need to consult the alignment information.

5.3 C evaluation

We used the SUIF compiler [8] to output instrumented C code. It acts like a C compiler with C as its target architecture. A disadvantage of this approach is that the instrumented C code does not capture stack references for function entry/exit, spill code and parameter passing. This will tend to underestimate the benefit of direct addressing as stack references provide many direct addressing opportunities, as quantified below in the Java evaluation.

The instrumented code has loops unrolled and is augmented with statistics gathering code. Every load and store in the program is analyzed and converted into a function call to our model. We verify at runtime that our static analysis was accurate.

5.4 C results

Figure 6 shows how many tag checks were eliminated for loads and stores for the Mediabench programs. From the number of tag checks eliminated, we computed the D-cache energy savings based on our extracted layout for the cache [15]. This model has tag search consuming 54% of a load and 43% of a store, broken down further into 10%/8% (load/store) for address bus, 25%/40% for data access, and 11%/9% for data bus.

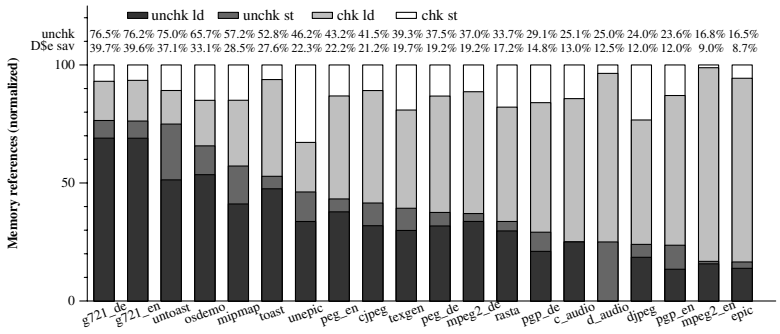

Fig. 6. Tag check elimination for Mediabench programs compiled by SUIF, using 8 DARs. The lowest part of the bar is tag unchecked loads, then unchecked stores. Over that are tag checked loads and stores. The number on top of each bar (unchk) is the percentage of tag checks eliminated. The number under that (D$e sav) is the percentage of dcache energy saved by eliminating the checks.

The results vary widely, with over 76% of checks eliminated for g721 decode (39.7% savings in data cache energy), down to 16.5% for epic. Direct addressing saves some energy on every application and even the small 8.7% energy savings on epic is likely to be larger than any overhead direct addressing introduces.

One reason for the spread is that some codes are more difficult to analyze, mostly due to pointer manipulation. One example is mpeg2 decode, for which the compiler was unsuccessful on the code as distributed with Mediabench. The code had one key loop which was manually unrolled, with a key matrix traversed in column-major order. By making four small edits to the source code to express the loop in a natural way, and to traverse the matrix in row-major order (which is also better for cache performance), the percentage of tag checks eliminated went from 6.2% to 37%.

Table 2 shows the data cache energy saved, and also the energy saving for the whole processor core including instruction and data caches. The energy consumption of the data cache relative to the entire core is highly dependent on the implementation. Our core design is highly optimized for low-power, consuming 100–250 pJ per instruction at 300 MHz in a 0.25 μm technology (<100 mW). For our design, we measured average data cache tag energy at 10% of the total core energy for Mediabench [15].

Benchmark	De^-	P+I +De^-	0off	8DAR lim	f()	r/w	# inst	# ld	# st	input
g721_de	39.7%	7.9%	11.6%	0.0%	0.0%	5.0	568719607	27155521	4567915	clinton.pcm
g721_en	39.6%	7.9%	12.0%	0.0%	0.0%	4.8	602714433	28470293	4557182	clinton.g721
untoast	37.1%	7.4%	2.7%	0.0%	0.0%	7.1	164673415	5211304	2741109	clinton.pcm
osdemo	33.1%	6.6%	6.0%	0.1%	0.2%	6.0	17768196	1005300	375520	out.ppm
mipmap	28.5%	5.7%	4.1%	0.0%	1.1%	5.6	50705063	3728696	1677708	out.ppm
toast	27.6%	5.5%	21.2%	28.0%	0.0%	11.6	325450576	33490458	4344146	clinton.gsm
unepic	22.3%	4.5%	40.8%	0.0%	0.0%	2.1	16471762	816031	676550	test_image.E
peg_en	22.2%	4.4%	61.4%	5.9%	0.0%	3.3	84217188	6191963	1415461	pgptest.plain
cjpeg	21.2%	4.2%	10.0%	1.9%	0.9%	5.9	35620933	2949426	758928	testimg.ppm
texgen	19.7%	3.9%	19.9%	2.1%	0.4%	2.2	146657184	9933559	3960265	out.ppm
peg_de	19.2%	3.8%	72.3%	0.2%	0.4%	2.9	46589722	3508783	818128	pegwit.enc
mpeg2_de	19.2%	3.8%	3.5%	0.0%	0.0%	2.7	270350477	19967230	3440148	mei16v2.m2v
rasta	17.2%	3.4%	35.5%	2.7%	0.0%	2.6	30132991	2866589	802709	map_weights.dat
pgp_de	14.8%	3.0%	70.6%	2.9%	0.1%	1.8	16299047	905321	287437	pgptext.pgp
c_audio	13.0%	2.6%	0.1%	0.0%	0.0%	13.7	18686936	443006	74056	clinton.pcm
d_audio	12.5%	2.5%	0.1%	0.0%	0.0%	13.7	17259137	369246	147816	clinton.adpcm
djpeg	12.0%	2.4%	14.4%	0.6%	0.3%	3.0	8882489	755752	305428	testimage.jpg
pgp_en	12.0%	2.4%	68.2%	2.0%	0.1%	1.8	28908438	1423749	428303	pgptest.plain
mpeg2_en	9.0%	1.8%	2.7%	0.0%	0.0%	5.6	3587002898	235379785	5349441	options.par
epic	8.7%	1.7%	24.7%	0.0%	0.0%	3.3	118204938	5971476	542458	test_image.pgm
average	21.4%	4.3%	24.1%							

Table 2. De^- is the data cache energy saved from eliminating tag checks. P+I+De^- is the energy saved for the processor plus instruction and data caches. 0off shows the percentage of tag unchecked accesses where the dominant and subordinate accesses were to the same address. f() shows how many tag checks happened as a result of function calls invalidating DARs. r/w gives the ratio of DAR reads to DAR writes. # inst gives the number of SUIF instructions executed by the benchmarks, and ld/st give the number of loads and stores.

The Table clearly shows the importance of offset information. While the results vary across benchmarks, most of the benefit of the DARs is not just from the program reusing the same location (0off column).

Our initial experiments indicated that 8 DARs captured most direct addressing opportunities across a range of benchmarks. The 8DARlim column shows how many more tag checks could be eliminated with an unlimited number of DARs versus the 8 used for the rest of the results. We compute this number by emitting liveness information for DAR candidates and doing post-hoc optimal register allocation. Only toast is able to soak up many more tag checks with more registers (it can profitably use 44). Every benchmark could make use of at least two DARs. Random permutation of register numbers makes the interference of function calls very small, as seen in the f() column. Finally, we see that each DAR value written is usually reused several times (r/w column), sometimes over 13 times, but averaging around 2–3 times.

6 Java implementation

Java bytecodes are normally interpreted directly or fed to a just-in-time compiler, but instead we used the FLEX compiler to compile Java bytecodes to MIPS assembly code. Java-to-native compilation is a good alternative for low-power environments if Java's dynamic loading capabilities are not usually needed, as the code can be highly optimized for low energy consumption.

The FLEX implementation used the same dominance analysis and DAR allocation algorithms as the SUIF implementation. The following sections first describe how heap memory references are mapped onto cache lines for Java programs, and then discuss the results of our experiments.

6.1 Object identity in Java programs

Our approach to finding references to the same cache line is different in Java than it was in C. Java's type-safety and object-orientation means there is additional pointer information available to the compiler.

All memory for Java objects comes from the system allocator. We modify the memory allocator to ensure that small objects are never split across cache lines and that larger objects are always aligned to the start of a cache line. The compiler can then simply determine cache-line equivalence based on object type and member field offset. This determination is performed on a very low-level representation just prior to instruction selection, so even access to object header words (like the class descriptor and hashcode) are visible to this "cache-line equivalence" analysis. This modified allocation policy potentially introduces fragmentation, which the allocator could deal with, e.g., by tracking "holes" and filling them in with small objects.

This type-based analysis is very simple, but accounts for a large number of eliminated tag checks in strongly object-oriented benchmarks like jess or jack. For more traditionally coded benchmarks, such as compress, there is need for further cache-line equivalence analysis of indexed array operations.

As with the C implementation, loops are unrolled in Java to expose more direct addressing opportunities. The unrolling strategy in Java is simpler: each loop which mentions an array is unrolled C/E times, where C is the cache line length, and E is the element size of the array with the smallest elements in the loop. This may over-unroll some loops, but guarantees that almost all the direct addressing opportunities are exposed. If the first element accessed in the loop is not

cache-line aligned, extra checks are placed within the unrolled loop to catch cache-line boundary crossings.

To further expose direct addressing opportunities and improve performance, the FLEX compiler inlines small final methods.

6.2 Java evaluation

FLEX outputs the MIPS instruction extensions for direct addressing (Table 1). Due to the limited number of offset bits in the instruction encoding, some loads (that use the global pointer) take one instruction while some loads (to data that is further than 32 KB from any register) take two instructions. The GNU assembler was modified to accept these instructions, and our extended MIPS ISA simulator models the state of the DARs (with dynamic correctness checks of DAR use). The Java runtime is written in C, and was compiled with gcc 2.7.2 with a MIPS target. The runtime is linked with the assembled Java code to give a MIPS binary that is run on the simulator.

The Java garbage collector was disabled for all runs. The collector, like the runtime, is written in C. The collector moves large amounts of data in memory with exact knowledge of object size and alignment, and so we expect that it could make heavy use of direct addressing. Modifying the collector was beyond the scope of these experiments, but including the modified collector should only improve the relative benefits of direct addressing.

Instead of modifying the system memory allocator to ensure cache alignment of heap data, we instead used conventional malloc and modified our checking code to ensure that all references are to the same 32-byte block of memory regardless of alignment.

6.3 Java results

Table 3 shows the percentage of tag checks eliminated for Java SPECjvm98 programs. Unlike our C evaluation, we ran each Java binary on the detailed energy simulator [15] to get exact energy dissipation numbers (except for mpegaudio which ran for too long and was estimated at 10%, as with the C benchmarks). Data cache tag check energy consumption was computed to be almost exactly 10% for every benchmark except raytrace, which has many memory accesses, and dissipates 13% of its energy in data cache tag checks.

The nSP column shows how many of our eliminated tag checks are to non-stack memory accesses. Most of the stack accesses are function entry/exit, and these are easy for the compiler to transform. The data

Benchmark	ntag	D\$$e^-$	Te^-	nSP	0off	f()
jess	62.8%	31.0%	6.2%	12.6%	2.0%	1.3%
jack	58.2%	28.0%	6.1%	43.3%	15.9%	0.4%
raytrace	56.7%	27.6%	7.6%	4.7%	0.6%	0.1%
compress	53.4%	26.3%	5.5%	26.4%	4.8%	1.2%
db	51.8%	25.7%	5.5%	5.2%	2.1%	1.6%
mpegaudio	18.0%	9.3%	1.8%	50.3%	25.2%	1.2%

Table 3. All benchmarks were run with -s10, which is the middle sized spec input. ntag is the number of data cache tag accesses eliminated. D\$$e^-$ is the data cache energy saved from these eliminated tag checks. Te^- is the energy saved for the processor plus instruction and data cache. nSP is the percentage of memory references that were tag unchecked, but did not reference the stack. 0off is the percentage of tag checks eliminated whose dominant and subordinate reference were to the exact same address. f() is the percentage of tag checks caused by having a function call invalidate a live DAR.

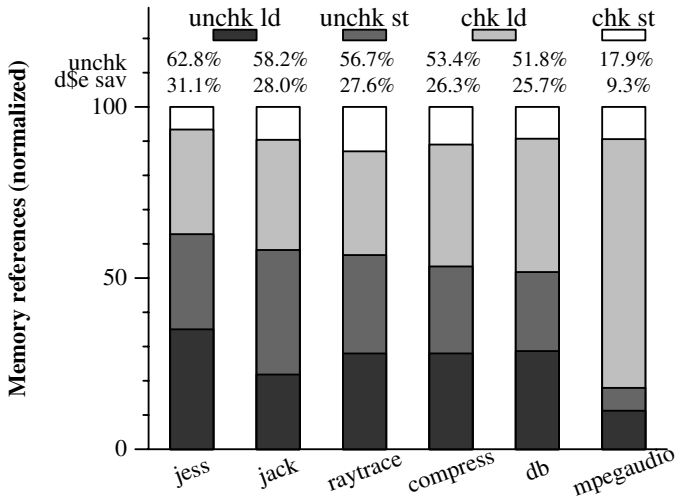

Fig. 7. Tag check elimination for SPECjvm98 programs compiled by FLEX using eight DARs.

for Java shows that stack references are about half (46–79%) of all memory references for SPECjvm98, and our analysis eliminates 67–82% of tag checks for these references. This gives an indication of the expected improvement if stack accesses were included in the SUIF C evaluation.

Table 3, like Table 2, shows the necessity of offset information. The number of zero offset references (where the dominant and subordinate access are to the same location) is lower in Java than in C because much of the tag check elimination comes from stack accesses on function entry and exit. These accesses load or store registers to sequential locations on the stack.

The f() column is the percentage of accesses that have to be tag checked because a function call between a DAR definition and use invalidated the DAR. As with our C benchmarks, random permutation of DAR numbers keeps this interference low.

Finally in Table 4, the mSP column shows that by ignoring spill code and parameter accesses, we are not missing a major opportunity. The generally low numbers indicate that the register allocator is not doing excessive spilling.

Mpegaudio sticks out because there is excessive spilling in this benchmark. Transforming the spill code to use direct addressing would get us a large part of the 52.0% of stack references which are not being analyzed. This would bring mpegaudio into the 50–60% tag elimination range of the other applications.

In order to transform spill code, we would generalize our direct register analysis and allocation to work on the post-register allocated version of the program (all the needed information is still available in FLEX).

Benchmark	Jinst	Jrefs	JavaSP	RunSP	mSP	# inst	# ld	# st
jess	44.6%	45.9%	66.2%	59.1%	0.5%	386362871	74217394	38927873
jack	60.0%	51.6%	45.2%	55.1%	4.4%	742795569	97902751	83399493
raytrace	19.1%	12.8%	79.7%	26.2%	6.9%	711506624	121545307	87011062
compress	99.7%	99.5%	49.3%	11.4%	1.8%	1995067192	318765365	182481314
db	63.7%	49.2%	53.9%	55.2%	0.2%	229082873	36830775	17634446
mpegaudio	7.9%	4.6%	62.9%	24.4%	52.0%	3798725510	860533959	164886641

Table 4. All benchmarks were run with -s10, which is the middle sized spec input. Jinst is the percentage of instructions executed in Java code. The remainder executed in the runtime. Jrefs is the percentage of memory references issued in Java. JavaSP is the percentage of Java memory references that are to the stack. RunSP is the percentage of memory references made to the stack by the Java runtime. mSP is the maximum possible contribution to the tag unchecked references if we converted every remaining stack access–namely spill code and parameter access. # inst/ld/st are the numbers of instructions, loads and stores from the Java code, not including the runtime.

7 Comparison with hardware tag-check elimination schemes

In this section, we compare our direct addressing scheme for eliminating tag checks at compile time with dynamic hardware alternatives that are invisible to software. One approach is for the hardware to remember the tag of the last cache line that was accessed and to compare this against the tag of the next memory access before enabling the tag search [2]. The main disadvantage of this scheme is that it adds a wide tag compare into the critical path of every cache access, adding several gate delays to this latency-sensitive path. A variant of this scheme is to remember the last line accessed within each cache subbank, and only power up cache tags if a different line is accessed within each subbank.

Table 5 compares results for the C and Java benchmarks using these two schemes. Using 8 DARs usually removes more tag checks than a hardware single last line buffer without the additional access latency, although with `pgp` the hardware scheme is significantly better. The hardware and software techniques can be combined, with the last line buffer used in cases where the DARs were not specified or unsuccessful. In this case, accesses will incur the additional cache access latency of the hardware scheme. The results in the fourth column of Table 5 show that combining the techniques usually does better than using each alone, indicating that they are capturing different types of cache line reuse.

The fifth column in Table 5 shows the results for the per-subbank last line buffer (16 subbanks). This removes many more tag checks than the previous schemes, but requires an extra tag comparator in each subbank and incurs the additional memory access latency. Finally, the sixth column shows the effect of adding 8 DARs to the per-subbank last line buffers. Here, there is little additional benefit (except for `mipmap`) as the hardware scheme has captured most of the available cache line reference locality.

The results for the Java benchmarks are similar, with the hardware last line scheme eliminating roughly the same number of tag checks as the 8 DAR scheme, but with the additional memory access latency. There is a smaller benefit to combining the hardware and software schemes for the Java programs, because the DARs only give benefit to the hardware schemes where the analysis was successful, as in jess and jack. Again, the per-subbank last line scheme performs well, removing 80–90% of all tag checks.

Program	8 DAR	last ln	last ln + 8 DAR	ll-sub	ll-sub + 8 DAR
C Benchmarks					
g721_de	76.5%	73.5%	82.1 +08.6%	98.4%	98.4 +00.0%
g721_en	76.3%	73.2%	81.7 +08.5%	98.4%	98.4 +00.0%
untoast	75.0%	39.6%	82.3 +42.7%	97.3%	97.5 +00.2%
osdemo	65.7%	47.8%	75.6 +27.8%	86.4%	88.4 +02.0%
mipmap	57.2%	22.5%	64.6 +42.1%	60.1%	85.7 +25.6%
toast	52.9%	15.0%	87.7 +72.7%	91.4%	98.5 +07.1%
unepic	46.2%	57.0%	71.2 +14.2%	81.0%	83.9 +02.9%
peg_en	43.2%	27.6%	46.6 +19.0%	65.5%	67.6 +02.1%
cjpeg	41.5%	17.5%	50.0 +32.5%	74.5%	79.6 +05.1%
texgen	39.3%	36.4%	56.2 +19.8%	78.4%	83.7 +05.3%
peg_de	37.5%	19.4%	41.2 +21.8%	59.0%	62.0 +03.0%
mpeg2de	37.1%	7.7%	40.4 +32.7%	84.9%	86.1 +01.2%
rasta	33.7%	19.1%	43.9 +24.8%	81.0%	85.8 +04.8%
pgp_de	29.2%	46.1%	57.2 +11.1%	89.8%	91.4 +01.6%
c_audio	25.1%	0.1%	25.1 +25.0%	65.1%	68.8 +03.7%
d_audio	25.1%	0.1%	25.1 +25.0%	58.1%	61.7 +03.6%
djpeg	24.0%	17.2%	30.5 +13.3%	64.5%	67.5 +03.0%
pgp_en	23.6%	54.5%	67.6 +13.1%	95.4%	96.8 +01.4%
mpeg2en	16.8%	7.9%	22.1 +14.2%	88.7%	89.6 +00.9%
epic	16.5%	8.9%	19.1 +10.2%	70.7%	72.4 +01.7%
Java Benchmarks					
jack	58.2%	54.6%	66.0 +11.4%	88.9%	90.7 +01.8%
raytrace	56.7%	66.9%	68.2 +01.3%	90.3%	90.7 +00.4%
compress	53.4%	54.8%	61.6 +06.8%	80.9%	82.3 +01.4%
jess	62.8%	58.2%	73.6 +15.4%	84.2%	87.7 +03.5%
db	51.8%	50.0%	62.5 +12.5%	81.2%	83.9 +02.7%
mpgaudio	18.0%	27.8%	36.5 +08.7%	81.4%	83.5 +02.1%

Table 5. Tag checks eliminated by 8 direct address registers (DARs), by a last line hardware tag compare (last ln), by adding 8 DARs to a single line buffer, by per-subbank last line buffers (ll-sub) with 16 subbanks, and by adding 8 DARs to the subbank last line buffers. The hardware last line schemes add the latency of an additional tag compare to all memory operations.

8 Related work

The ARM instruction set includes load/store multiple instructions that can be used to avoid tag checks for sequential accesses to the same cache line [18]. These instructions are typically only used for procedure call/return, whereas our model allows significantly greater flexibility. For example, the results we presented for the C Mediabench code were for non-stack accesses which are much less amenable to load/store multiple.

Some researchers [2, 14] have described hardware L0 caches designed for low power access. These schemes have performance impacts, whereas the direct addressing scheme does not affect performance. Direct addressing can also be combined with some of these hardware schemes to save further power.

Other researchers [4, 9, 17] have developed software caching schemes that use compile-time information to reduce software tag checks. FlexCache [17] adds HotPage registers, which are similar to DARs except they also hold a tag along with the direct address. They are used as a small compiler-managed hardware tag array for a software associative cache. The HotPage-likely compiler analysis implements static software way-prediction to index the likely HotPage register holding the translation for a given memory access. The speculation is checked by a hardware compare of the virtual address with the HotPage tag. The authors mention that an additional optimization, HotPage-predictable analysis [4], could avoid this tag check but do not include compiler algorithms or results. In contrast, our work removes tag checks from a hardware associative cache scheme with no performance penalty, and our compiler analysis avoids tag checks by statically guaranteeing two accesses are to the same line.

Fisher [12] and Ellis [3] were the first to use loop unrolling to improve the alignment of memory references in a loop body. Their work was done in the context of a clustered VLIW in which main memory was divided among separate banks. Their architecture supported a fast path to memory when data were located on a cluster's local memory bank. Alignment of memory operations was therefore an important factor in machine performance.

Barua et al. expanded on these ideas and introduced Modulo Unrolling [1]. This work introduced precise equations for determining the unroll factors for loop nests. In Modulo Unrolling, outer loops may be unrolled to create aligned references outside the inner loop. This work was done in the conext of the RAW machine [5] in which processor memory is distributed across processing tiles. As is the case with the

clustered VLIW, access to a local bank is faster than access to a remote bank.

9 Conclusions

Direct addressed caches provide a new hardware-software interface to use energy of cache accesses. Direct addressing uses compile-time information plus a minimal amount of hardware to remove data cache tag checks, thus saving energy. Our implementations of direct addressing in a C and Java compiler resulted in data cache energy savings from 9–40% for C and 9–31% for Java. In contrast to other cache energy saving techniques, direct addressing does not change the performance of the processor, it just reduces the amount of microarchitectural work the processor performs.

10 Acknowledgements

This work was funded by DARPA PAC/C award F30602-00-2-0562, NSF Grant CCR-0073510, DARPA/AFRL Contract F33615-00-C-1692, NSF Grant CCR00-86154, NSF Grant CCR00-63513, and NSF CAREER Grant CCR-0093354.

References

1. Rajeev Barua, Walter Lee, Saman Amarasinghe, and Anant Agarwal. Memory Bank Disambiguation using Modulo Unrolling for RAW Machines. In *Proceedings of the Fifth International Conference on High Performance Computing*, Chennai, India, Dec 1998.
2. T. Burd. *Energy-Efficient Processor System Design*. PhD thesis, University of California at Berkeley, 2001.
3. J. R. Ellis. *Bulldog: A Compiler for VLIW Architectures*. The MIT Press, Cambridge, Massachussetts, 1985.
4. C. A. Moritz *et al.* Hot pages: Software caching for RAW microprocessors. *MIT-LCS Technical Memo LCS-TM-599*, August 1999.
5. Elliot Waingold *et al.* Baring It All to Software: RAW Machines. 30(9):86–93, Sep 1997.
6. J. Montanaro *et al.* A 160-MHz, 32-b, 0.5-W CMOS RISC microprocessor. *IEEE JSSC*, 31(11):1703–1714, November 1996.
7. M. Rinard *et al.* The FLEX compiler infrastructure. 1999–2001. `http://www.flex-compiler.lcs.mit.edu`.
8. M. S. Lam *et al.* The SUIF compiler system. 1992–2001. `http://www-suif.stanford.edu`.

9. O.S. Unsal *et al.* Cool-Cache for hot multimedia. In *MICRO–34*, 2001.
10. S. B. Furber *et al.* ARM3 - 32b RISC processor with 4kbyte on-chip cache. In G. Musgrave and U. Lauther, editors, *Proceedings IFIP TC 10/WG 10.5 Int. Conf. on VLSI (VLSI'89)*, pages 35–44. Elsevier (North Holland), 1989.
11. W. Lee *et al.* Space-time scheduling of instruction-level parallelism on a RAW machine. *ACM SIGPLAN Notices*, 33(11):46–57, 1998.
12. Joseph A. Fisher, John R. Ellis, John C. Ruttenberg, and Alexandru Nicolau. Parallel Processing: A Smart Compiler and a Dumb Machine. In *ACM SIGPLAN '84 Symposium on Compiler Construction*, pages 37–47, June 1984.
13. Intel Corp. *Intel Xscale core developers manual*, order no. 273473-001 edition, December 2000.
14. J. Kin, M. Gupta, and W. H. Mangione-Smith. The filter cache: An energy efficient memory structure. In *MICRO–30*, pages 184–193, 1997.
15. R. Krashinsky. Microprocessor energy characterization and optimization through fast, accurate, and flexible simulation. Master's thesis, Massachusetts Institute of Technology, May 2001.
16. A. Ma, M. Zhang, and K. Asanović. Way memoization to reduce fetch energy in instruction caches. *ISCA Workshop on Complexity Effective Design*, July 2001.
17. C. A. Moritz, M. Frank, and S. Amarasinghe. Flexcache: A framework for flexible compiler generated data caching. *Lecture Notes of Computer Science, Springer–Verlag*, 2001.
18. M. Muller. Power efficiency & low cost: The ARM6 family. In *Hot Chips IV*, August 1992.
19. R. Panwar and D. Rennels. Reducing the frequency of tag compares for low power I-cache design. In *SLPE*, pages 57–62, October 1995.
20. M. Zhang and K. Asanović. Highly-associative caches for low-power processors. *Kool Chips Workshop, MICRO-33*, 2000.