# Link-Time Optimization for Power Efficiency in a Tagless Instruction Cache


Timothy M. Jones
School of Informatics
University of Edinburgh
United Kingdom
tjones1@inf.ed.ac.uk

Sandro Bartolini
Faculty of Engineering
University of Siena
Italy
bartolini@dii.unisi.it

Jonas Maebe
ELIS Department
Ghent University
Belgium
jonas.maebe@elis.ugent.be

Dominique Chanet
Gateway Architecture Group
Technicolor
Belgium
dominique.chanet@technicolor.com



***Abstract*—The instruction cache is a critical component in any microprocessor. It must have high performance to enable fetching of instructions on every cycle. However, current designs waste a large amount of energy on each access as tags and data banks from all cache ways are consulted in parallel to fetch the correct instructions as quickly as possible. Existing approaches to reduce this overhead remove unnecessary accesses to the data banks or to the ways that are not likely to hit. However, tag banks still need to be checked.**

**This paper considers a new hybrid *hardware* and *linker-assisted* approach to tagless instruction caching. Our novel cache architecture, supported by the compilation toolchain, removes the need for tag checks entirely for the majority of cache accesses. The linker places frequently-executed instructions in specific program regions that are then mapped into the cache without the need for tag checks. This requires minor hardware modifications, no ISA changes and works across cache configurations. Our approach keeps the software and hardware independent, resulting in both backward and forward compatibility.**

**Evaluation on a superscalar processor with and without SMT support shows power savings of 66% within the instruction cache with no loss of performance. This translates to a 49% saving when considering the combined power of the instruction cache and translation lookaside buffer, which is involved in managing our tagless scheme.**


## I. Introduction

Power efficiency is a major concern for all types of computing system, from embedded devices through to high-performance servers. Power is a first class design constraint, as processor manufacturers seek to reduce chip packaging and cooling costs and increase battery lifetimes. However, reduced power consumption must be carefully balanced against performance impacts in order to achieve truly energy-efficient systems.

The instruction cache is a critical component of any processor, needing to be power efficient while maintaining high performance. Furthermore, caches are energy hot spots, meaning that it is important to tackle their power consumption to avoid high local temperatures [18].

Despite this, current cache designs waste significant amounts of energy by accessing tag and data banks from all ways in parallel, in order to keep cycle times low. Existing approaches have tackled this problem using a variety of hardware-only techniques [13], [21], [23]. However, these schemes depend on hardware prediction and do not make use of the compilation toolchain's detailed knowledge.

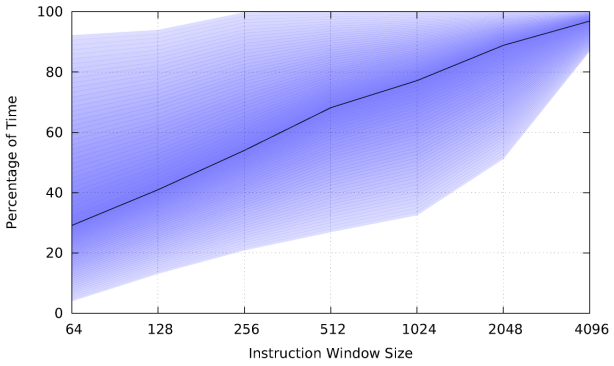



Fig. 1. Fraction of time that an instruction can be found within the last $n$ executed instructions for varying $n$. We show the maximum, average and minimum across the SPEC CINT2000 benchmarks. Instructions are found within the window 77% of the time when $n = 1024$ (the size of a memory page).

This paper takes a different approach. Figure 1 shows the fraction of time that an instruction can be found within the previous $n$ executed instructions, for varying values of $n$. We show the average line and the area between the maximum and minimum across our benchmarks (described in section IV). As can be seen, there is a high degree of temporal locality between the application's instructions. By exploiting this information, allowing the hardware and software to work in tandem, we can save more instruction cache power with less reliance on hardware prediction mechanisms.

We propose a novel profile-guided link-time optimization to select frequently-executed application code and place it in specific regions of the program binary (called *tagless regions*). We then add modest hardware extensions to the instruction cache and memory management unit to take advantage of this layout. The hot code is accessed without tag checks, significantly reducing instruction cache power. This process maps the temporal locality in figure 1 to spatial locality by placing the frequently-accessed code together.

Our scheme is extremely versatile, working across cache configurations, and requires no ISA changes. It can be implemented with only minor modifications to the structure

| Phase | Input | Extra Constraints | Output |
|---|---|---|---|
| Create initial working set | Set $F$ of most frequently-executed basic blocks | None | Clusters $C_{1..c}$ |
| Reduce incoming edge weight | Clusters $C_{1..c}$ and set $B$ of unused basic blocks | Reduce cluster's total incoming edge weight | Clusters $C'_{1..c}$ |
| Consolidate clusters | Clusters $C'_{1..c}$ | None | Clusters $C''_{1..d}$ |
| Create tagless regions | Clusters $C''_{1..d}$ | Cluster execution weight per incoming edge must exceed threshold $h$ | Tagless regions $T_{1..n}$ |
| Fill out tagless regions | Tagless regions $T_{1..n}$ and set $B'$ of unused basic blocks | Maintain or reduce region's total incoming edge weight | Tagless regions $T'_{1..n}$ |

Fig. 2. Overview of the tagless region formation algorithm.

of a traditional set-associative cache. The novelty in our approach is the ability to use the cache with and without tag checks through use of the linker for support. Furthermore, our hardware and linker modifications are independent. Therefore our innovative cache architecture can run unmodified binaries and our optimized programs can be executed on processors lacking support for tagless instruction caching. The absence of dependences across the software/hardware interface enhances the flexibility of our approach.

We evaluate our schemes on a superscalar processors with and without multi-threading support. Our technique achieves 49% dynamic power savings in the instruction cache and translation lookaside buffer with no performance loss.

The rest of this paper is structured as follows. Section II explains our linker pass to place frequently-executed code in tagless regions. Section III then describes the hardware modifications needed to support tagless instruction caching. Section IV describes our experimental setup and section V presents our results. We describe related work in section VI and finally, section VII concludes.

## II. Program Analysis & Rewriting

This section describes the program analysis and rewriting needed to support tagless instruction caching. Our technique uses profile information to identify frequently-executed basic blocks, which are then grouped together according to the control flow graph to form tagless program regions. We gather the execution count of each edge between basic blocks within the control flow graph. We then lay out the program using the linker, as it is the only compilation tool that knows which and where all code ends up in a program.

We use a tagless region size of one memory page that corresponds to 1024 instructions. This simplifies our hardware extensions (described in section III). Furthermore, as figure 1 shows, for 77% of the time on average (and 99% of the time for the best benchmark) the next instruction can be found within this window.

### A. Tagless Region Formation

The intuition behind our tagless region formation algorithm and our goals are as follows:

1) We want to spend as much time as possible executing instructions from a tagless region. To enable this we first choose basic blocks with a high execution count over those with a lower frequency.
2) We want to minimize switching amongst tagless regions, or to tagged code, as this reduces efficiency (see section III-C). Hence, we try to keep connected basic blocks in the same tagless region.
3) We want to fill the tagless regions as much as possible to maximize the tagless accessing. We therefore finalize the regions using leftover basic blocks that do not negatively impact the two other goals.

A schematic overview of the algorithm is displayed in figure 2. Note that every phase operates under the additional constraint that no cluster can exceed the size of a single tagless region at any time.

We start by creating a set $F$ that contains the most frequently-executed basic blocks. The total size of the instructions in $F$ is no greater than the size of all $n$ tagless regions that we wish to fill. This initial restriction allows us to concentrate on the most frequently-executed basic blocks, to ensure that we meet goal 1 above.

From this set $F$, we create an initial working set of clusters of connected basic blocks. We iteratively remove the most frequently-executed basic block from $F$ and place it in a new cluster $C_i$. We then expand the cluster by adding blocks from $F$ along the most frequently-executed edges into and out of the cluster.

During the second phase we optimize clusters $C_{1..c}$ by adding free basic blocks (from set $B$) that reduce the *incoming edge weight*, resulting in clusters $C'_{1..c}$. This weight is the sum of the execution counts of all edges that transfer control into the cluster. Minimizing this metric helps to meet goal 2 above. Blocks added here include, for example, paths of if-then-else statements that are infrequently taken.

The third phase consolidates the clusters. The previous phase may have connected more clusters to each other. We combine these clusters and obtain a new set $C''_{1..d}$, working towards goal 2.

We now have to fit our collection of clusters $C''_{1..d}$ into the tagless regions $T_{1..n}$ during phase four. We first sort the clusters based on their *instruction execution weight* (iew) ratio. This ratio represents the number of dynamic instructions in the cluster divided by the total execution count of the incoming edges. The higher the ratio, the more instructions are executed each time the cluster is entered. Clusters whose *iew* ratio is below a specified threshold $h$ are discarded. Once again, this satisfies goals 1 and 2. We have found, experimentally, that a

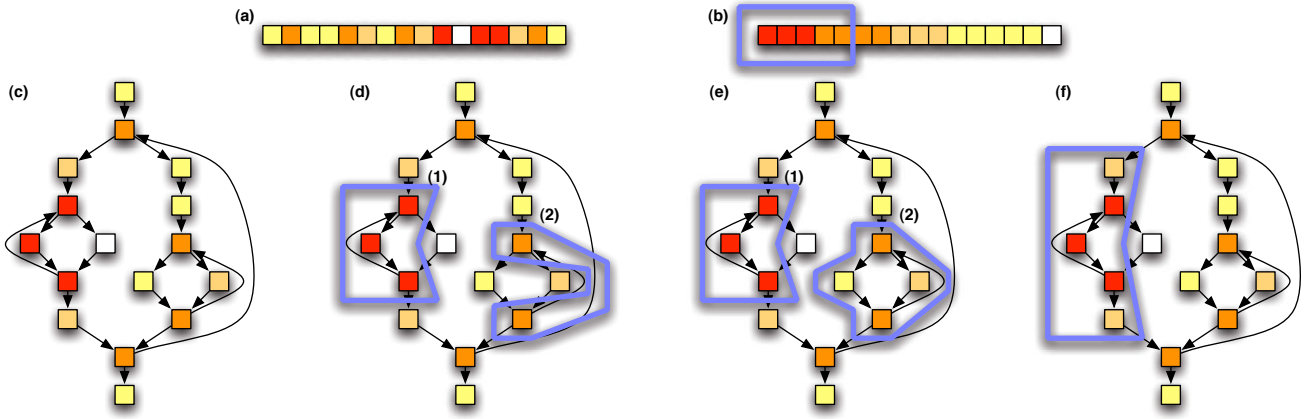



Fig. 3. Example of our code layout scheme. In (a) and (b) we show the original program split into basic blocks, in the latter case ordered by decreasing execution frequency. Steps (c) to (f) show how the tagless regions are constructed based on the program's control flow graph.

threshold of 100 is a good cutoff point to discard unprofitable clusters. Next, we place the remaining clusters into the tagless regions using a "first fit" approach.

We finalize the regions in phase five by adding, as space permits, executed basic blocks that do not cause an increase in the cluster's incoming edge weight, so as to satisfy goal 3 without hurting goal 2. Such basic blocks may still be found due to our discarding of clusters during phase four.

### B. Example

Figure 3 shows an example of our linker pass. Step (a) shows the original program, with each basic block represented using a square. The shade of the square indicates the execution count of the basic block obtained through profiling. The white squares represent blocks that were not executed at all during this profiling run.

In step (b) we sort all basic blocks in order of execution count. We then create our set of the most frequently-executed blocks, $F$. The size of $F$ is equal to the combined size of all $n$ tagless regions. In this example we assume that $n = 1$ and that the first five basic blocks are enough to fill this space.

Step (c) displays the control flow graph (CFG) of the program. We make the first cluster by taking the most frequently-executed block from $F$ and iteratively adding basic blocks when they are connected via edges in the CFG. Once this cluster is finished, we repeat until $F$ is empty. This produces the two clusters shown in step (d).

Step (e) grows both clusters using blocks that decrease the total execution count of their incoming edges (edge weight). The white block is not added to cluster (1) because it was not executed while profiling. The dark block at the bottom is not added to cluster (2) because the incoming edge from the left-hand side would increase its edge weight.

The clusters are now sorted based on their execution ratios, i.e., the ratio of their total dynamic instruction count to their total dynamic incoming edge count. We assume that cluster (1) has the highest ratio and add it to the tagless region. Clusters (1) and (2) together do not fit in the tagless region, so cluster (2) is discarded.

Step (f) performs a final enlargement of the code in the tagless region. Here we select basic blocks that reduce or maintain the incoming edge weight. Finally, we pad the tagless region until it is full.

### C. Summary

This section has presented our region formation algorithm. The linker uses an edge profile to create clusters of frequently-executed basic blocks. Then, it places these clusters in tagless program regions, maximizing the execution time spent within these regions and minimizing the switching between regions. The next section describes our hardware modifications that take advantage of this layout to provide tagless instruction caching.

## III. Tagless Instruction Caching

This section presents our approach to tagless instruction caching, detailing the microarchitectural changes and the operation of the cache. We begin with an overview of how the scheme works.

### A. Overview

Our instruction cache has two modes of operation: tagged and tagless. When we are fetching instructions in tagless mode, energy is saved because we omit all tag checks and only access one data bank from the cache. The page descriptors in the memory management unit contain extra bits to indicate whether the page contains a tagless region.

Figure 4 gives an overview of an instruction fetch in both tagged and tagless modes for a 2-way set-associative cache. When fetching an instruction in tagged mode the required address is routed to tag and data banks for all cache ways, i.e., light and dark boxes in figure 4. Tags and data are read out of all arrays and, on a hit, the data from the selected way is sent back to the processor. However, energy is wasted reading data from the ways that don't contain the required instruction. Although checking the tag and data banks sequentially could reduce this problem [12], the cache hit time would be significantly increased.

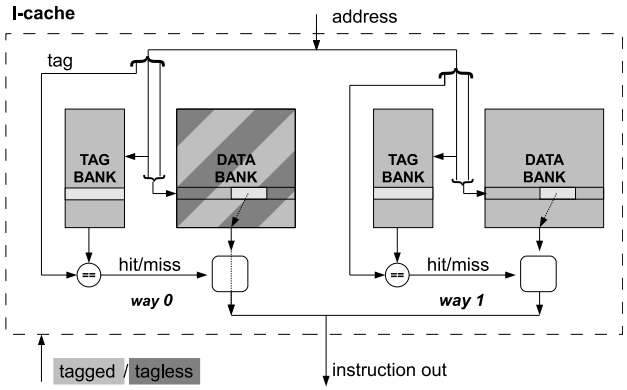



Fig. 4. Internal modules accessed during a read operation on a 2-way associative cache in case of tagged and tagless access modes. When in tagged mode, all tag and data banks are accessed (in light gray), while in tagless mode, only one data bank is accessed (in dark gray), thus reducing power.

The key observation behind our tagless caching scheme is that, if we can *guarantee* that the instruction already is in the cache at a particular location, energy is also wasted in checking all tags. When the cache operates in tagless mode, the instruction translation lookaside buffer (ITLB) determines the way that each tagless region resides in, meaning that they are in one specific location only. The address of each instruction is used to determine the line it is in as normal. In the example shown in figure 4, the instruction is already in the cache, so it is read out without performing any tag checks and accessing just one data bank, indicated in dark gray. By doing this, only a fraction of the full access energy is consumed.

Section III-B explains how we guarantee that instructions are in the cache when accessing in tagless mode and section III-C explains how the ITLB assigns a way to each tagless region.

### B. Filling The Cache

The ability to use the cache in tagless mode relies on our scheme *guaranteeing* that the correct instructions reside in the cache in a specific location given by their address. Since there are no tag checks when operating in this mode, there are also no cache misses. We cannot, therefore, rely on the normal cache miss handler to bring in instructions if they are not already there.

We solve this problem through the use of an additional *tagless valid* bit (tv-bit) for each cache line. As shown in figure 5, this bit indicates that its corresponding line contains instructions from a tagless region. Whenever an instruction is fetched in tagless mode, a specific cache line is accessed in parallel with its associated tv-bit. If the tv-bit is set, then the line is valid for use in tagless mode. If it is unset then a normal miss operation is started to bring the required instructions into the cache. Section III-C describes how we support multiple tagless regions where several addresses map to the same cache line and tv-bit.

The tv-bits are set and reset when servicing a miss in tagless and tagged mode, respectively. Initially they are all reset to 0. The first time that an instruction from a tagless region is fetched, the access will fail and the line will be brought in from the next level of memory. The tv-bit is set to show that the line contains valid tagless instructions and fetch can continue. We do not need to write the tag for this address, saving more energy, but must reset the standard valid bit for the line so that a miss occurs if a tagged access occurs. We ignore this when accessing in tagless mode.

The only change to accesses in tagged mode is that the tv-bit for the replaced line must be reset on a miss so that the processor knows that the line no longer contains instructions from a tagless region. Setting and resetting the tv-bits in this manner means that we can support tagless regions and tagged code at the same time in the cache on a line-by-line and on-demand basis. As a result, we load only the required lines from tagless regions.

We assume each way of the cache is placed in its own independent bank. In tagless mode we simply ignore the tag bits and only access the data bank in the way provided by the ITLB (described in section III-C). The other data banks (and all tag arrays) can be gated to avoid consuming any dynamic energy.

Using this scheme, the cache does not need to be aware of the size and number of regions in the executable. The tagless regions defined and prepared by the linker will map to the available tagless space present in the target cache. This allows different tagless caches to take advantage of the same tagless-optimized executable. There is no need for recompilation to adapt to different targets and to new generations of the same processor (e.g., a processor core with a larger cache, or tagless support capabilities).

### C. ITLB Support

As described in the previous section, the cache's mode of operation depends on whether instructions are being fetched from a tagless region or not. To keep the hardware overheads to a minimum, we make each tagless region the size of a memory page, motivated by figure 1.

We augment each entry of the ITLB with information describing whether the page is currently present within the cache and the way that it is mapped to. Figure 5 shows how the ITLB is modified for a 32kB, 4-way cache with 4kB pages. We add a single bit to each ITLB entry to indicate whether the page contains a tagless region (the T bit). We also provide a further two bits to indicate the way that the page is mapped to, if tagless (Rway). Finally, to identify stale way information we also include a region bit (R) and a valid bit (V). The way information is required to avoid conflicts between tagless pages that are mapped into the cache. To perform way assignment he ITLB maintains an LRU chain of cache ways for each cache region. In figure 5 this corresponds to two chains, since the cache has eight regions split into four ways.

When a new page containing a tagless program region is brought into the ITLB, its T, R and V bits are set and its Rway is set to the LRU way from the corresponding region's chain.

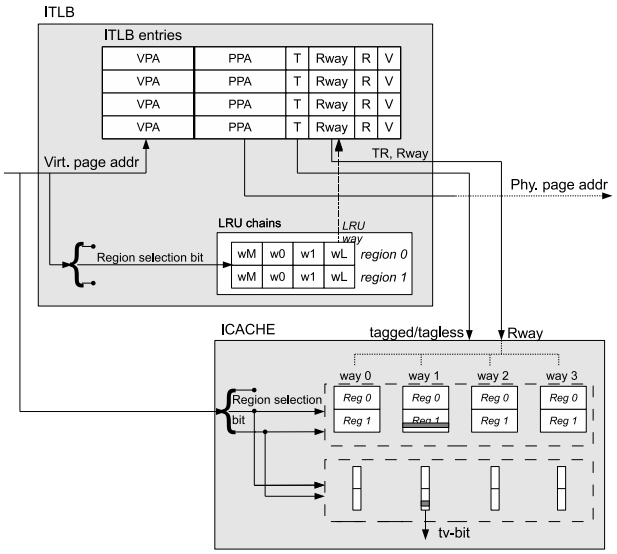



Fig. 5. ITLB operation and cache *tagless valid* bit (tv-bit). Each page descriptor contains a bit to identify pages from a tagless region (T), the way that the tagless pages are positioned in the cache (Rway), the region (R) and whether the page is valid in the cache (V). The ITLB maintains an LRU chain for each cache region to help avoid conflicts between tagless pages.

Any other page that has the same Rway and is region (R bit) has its mapping cleared by reseting its V bit. In addition, all tv-bits corresponding to this region and way are reset in the cache. This ensures that instructions are never fetched from the wrong tagless region.

Maintaining the Rway information within the ITLB ensures that the cache's tv-bits only need to be reset when control passes to a page that is not currently mapped into the cache. Transitions between pages that are already mapped incur no overheads and cause no conflicts. Conflicts between mapped tagless pages and tagged pages are dealt with by the tv-bits within the cache, as described in section III-B.

The cache and ITLB are accessed in parallel, so a page's T bit and Rway are unknown when starting the cache access. Hence, we speculatively use their previous values and replay the access if needed. This avoids increasing the access latency. Thanks to the linker pass, the majority of accesses are within the same page as the previous access, and therefore this speculation is rarely wrong. In fact, there two scenarios where we have to replay the access. First, when transitioning from tagless to tagged mode, since we haven't performed a tag check so won't have found the correct instruction. Second, when transiting between tagless pages that are mapped to different ways, since the way prediction will be incorrect. Both replays incur a single (cache) cycle performance penalty and the power of performing a redundant tagless access. When transitioning from tagged to tagless mode there is no need to replay the access since we have already checked all cache ways for the required data.

| Parameter | Configuration [SMT] |
|---|---|
| Width | 4 Instructions |
| ROB size | 128 [256] Entries |
| LQ/SQ size | 32/32 [64/64] Entries |
| Issue queue | 64 [128] Entries |
| Br predictor | Gshare |
| BTB | 4096 Entries |
| L1 Icache | 32kB 4-Way 64B Line, 2 Cycle Hit |
| L1 Dcache | 32kB 4-Way 64B Line, 2 Cycle Hit |
| Unified L2 cache | 2MB 8-Way 64B Line, 20/250 Cycle Hit/Miss |
| Int and FP RF | 128/128 [256/256] Entries |
| Int FUs (cycles) | 3 ALU (1), 1 Mul (3) |
| FP FUs (cycles) | 2 ALU (2), 1 MulDiv (4/12) |

TABLE I
THE PROCESSOR CONFIGURATIONS USED. PARAMETERS IN SQUARE BRACKETS DENOTE VALUES USED IN THE SMT PROCESSOR.

### D. Summary

This section has described our instruction cache and the additional hardware needed to support tagless instruction fetch. We augment each cache line with a tagless valid bit (tv-bit) that indicates whether it contains valid instructions from a tagless program region. We make each tagless region the size of a memory page and include extra bits within the ITLB to identify tagless regions, the way they are mapped to, and to prevent conflicts between them. Our hardware modifications are backwards-compatible and allow us to provide tagless instruction caching for any program binary.

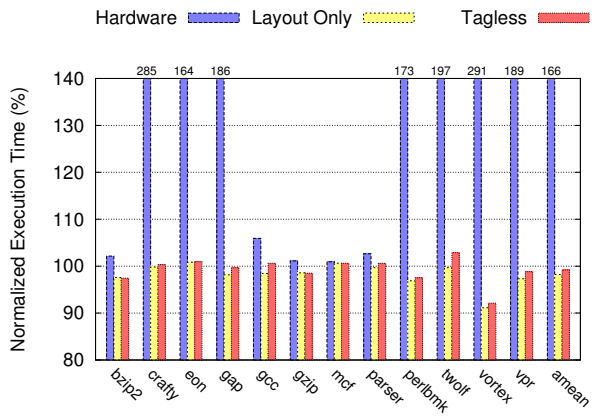
The next section presents our experimental setup and then section V evaluates our technique and the region formation algorithm over a variety of tagless region configurations.

## IV. EXPERIMENTAL SETUP

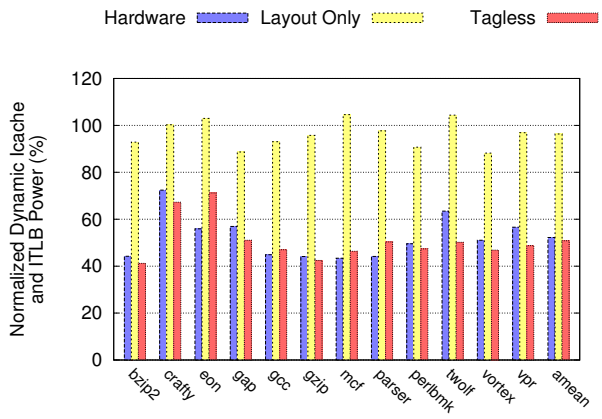
*a) Benchmarks and Compilation Toolchain:* For our experiments we compiled the SPEC CINT2000 suite [27] for a generic PowerPC architecture using GCC 4.2.0, with optimization level O2. For our SMT results we randomly picked 20 pairings of the benchmarks to run.

We wrote our code placement scheme in the Diablo [8] link-time rewriter. To extract the profile information needed by our linker pass we ran each benchmark with its *train* input set. For the performance and energy evaluations we used the *reference* input set. Our baseline binaries have also been relinked with Diablo, but omitting the pass that creates the tagless regions (i.e., with no basic block reordering). This ensures that the effects of our technique are solely due to our region formation algorithm and are not a facet of our compilation tools. We have also verified that the baseline power and performance measures are in line with those that do not use Diablo.

*b) Simulator:* We have evaluated our techniques on a high-performance out-of-order superscalar processor, with and without SMT support. The parameters are shown in table I. We implemented this within the M5 simulator [4]. We used McPAT [20] to model the power consumption of the whole processor. For our techniques, we also include the energy




(a) Performance




(b) Icache and ITLB dynamic power

Fig. 6. Results of our tagless accessing approach for each benchmark, normalized to the baseline. We also show the hardware and software elements of our scheme alone (Hardware and Layout Only), motivating the need for a hybrid solution.

overheads of the additional hardware required to support tagless instruction accessing.

To accurately simulate our schemes on realistic, high-performance workloads we used SMARTS [31] methodology. We simulated at least 10,000 samples from each benchmark, obtaining cycles per instruction and energy per instruction estimates to within $\pm 3\%$ with 99.7% confidence. For our SMT evaluations we fast-forwarded each workload for 1 billion instructions, then simulated at least 1 billion more for each benchmark. We evaluated power consumption in preference to energy to minimize the effects of the performance gains achieved by some benchmarks. Programs that run faster than the baseline must save comparatively more energy in order to achieve power savings.

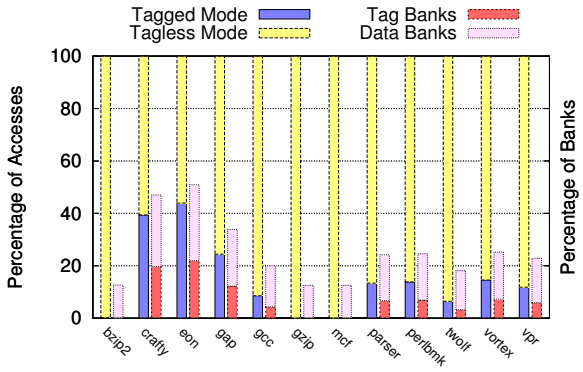



Fig. 7. Breakdown of cache and bank access types.

## V. Results

This section presents the results of our tagless instruction caching technique. We first evaluate the two components of our approach separately and combined. Then we perform a comparison against alternative power saving schemes. Finally we look at the impact on a shared instruction cache within an SMT processor.

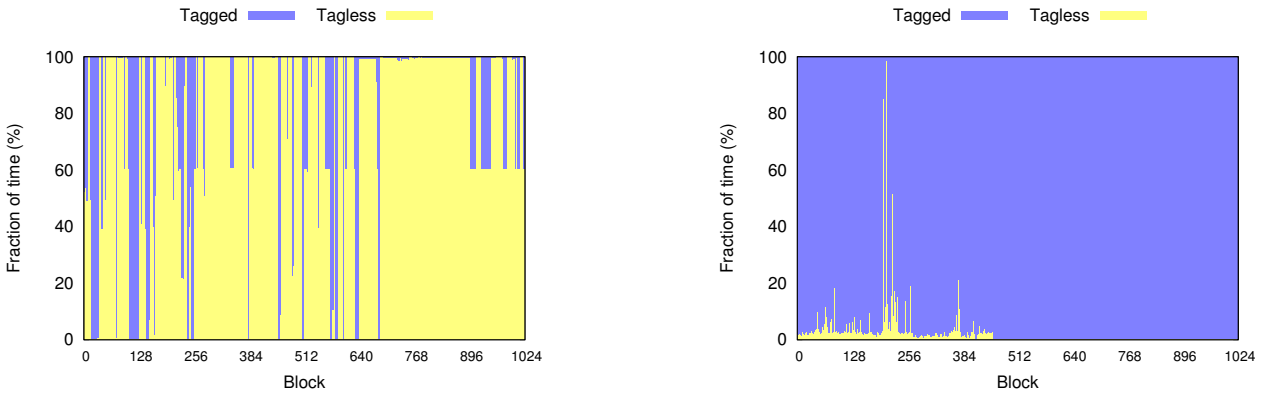
### A. Initial Results

We first show the results of our approach on performance and power, then analyze the instruction cache access patterns that cause the power savings achieved. Figure 6 shows our scheme and its two component parts normalized to the results of running an unmodified binary on the baseline architecture. We show our hardware technique alone (i.e., unmodified binaries accessing all code in tagless mode), our code layout approach (i.e., our relinked binaries on the baseline architecture), and our hybrid tagless scheme.

*c) Performance:* What is clear from the performance results (shown in figure 6a) is that using only our hardware mechanism alone is not viable. Seven out of our twelve benchmarks experience severe slowdowns with an average performance loss of 66%. This is due to conflicts between tagless regions within the cache causing a whole region's tv-bits to be cleared, as described in section III-C. There are over 250,000 clears of region tv-bits in *vortex* alone and this leads to *Hardware*'s large performance losses.

In contrast there is little change in performance from the baseline when using our code layout approach in isolation. For some benchmarks there is a slight increase in performance (e.g., *vortex* which is 9% faster). This is caused by an increased branch predictor accuracy, meaning that the processor dispatches fewer instructions down mispredicted program paths. Our linker pass groups basic blocks together and adds branch instructions in order to keep the correct control flow through the program. This can make the control flow more regular and easier to predict.

The results from using our hybrid tagless scheme are similar to the code layout only approach although, in general, the code layout scheme is marginally faster. This is due to the hybrid technique needing to reply cache accesses occasionally when the T or Rway information is mispredicted. However, this is infrequent and on average our hybrid scheme is a negligible 1% faster than the baseline, with only *twolf* showing a 3%




(a) Bzip2

(b) Crafty

Fig. 8. Fraction of time each cache line holds tagged or tagless instructions for two benchmarks. In bzip2 the cache mainly holds tagless instructions. The opposite is true for crafty, where few lines contain tagless instructions for a signficant fraction of time.

slowdown due to a higher than average misprediction rate for the T and Rway information.

*d) Power:* We next consider the effects of each of these approaches on the dynamic power consumed by the instruction cache and ITLB, shown in figure 6b. Although *Hardware* provides good savings, this is offset by the performance impact that it causes. *Layout Only*, on the other hand, neither causes significant power savings nor losses and has an average of 4% power savings. This confirms that the tagless layout is backwards compatible with standard caches and causes no additional overheads.

This figure shows that our hybrid approach can achieve significant power savings across the suite of applications. In particular, *bzip2*, *gcc*, *gzip*, *mcf*, *parser*, *perlbmk*, *twolf*, *vortex*, and *vpr* all achieve savings of at least 50%. Even the worst benchmark, *crafty* at 67% power, still achieves savings of 33%. Across the board, the average is 49% power savings for our hybrid tagless scheme. This corresponds to 66% savings in the instruction cache's dynamic power alone.

*e) Analysis:* Having shown the benefits of tagless instruction caching we now consider the reasons for the power savings achieved. Figures 7 and 8 shows our analysis.

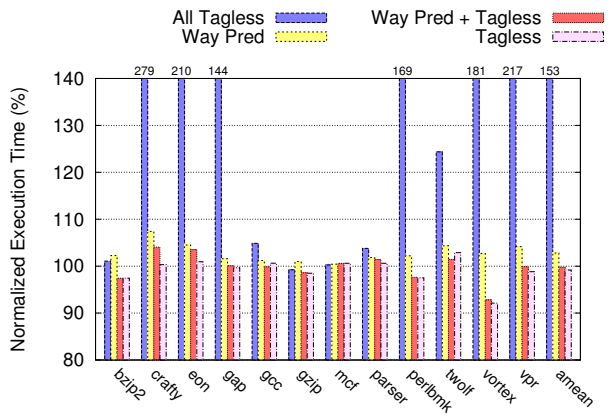
In figure 7 the fraction of instruction cache accesses each program makes in tagless and tagged modes are shown. It also shows the proportion of tag and data banks that are accessed, normalized to the number that would be accessed usually. Figures 8a and 8b show the fraction of time each cache line holds instructions from a tagless page or a tagged page for *bzip2* and *crafty*, respectively. We picked these benchmarks in particular because they lie at opposite ends of the spectrum. We count a cache line as containing tagless instructions whenver its tv-bit is set, and tagless instructions otherwise.

As can be expected, there is a clear correlation between accesses in tagged mode and the number of tag and banks accessed. For example, *bzip2* accesses the instruction cache in tagless mode almost all the time. Therefore, almost no tag banks ever need to be consulted and only one data bank is accessed on each fetch. Figure 7 shows that only 12.5% of the instruction cache banks are accessed using this scheme. Furthermore, figure 8a shows that for the majority of the time, cache lines hold only tagless instructions. The linker is able to place the vast majority of this program's frequently-accessed code into tagless regions, meaning that almost all accesses use the low-power tagless cache mode, allowing *bzip2* to reduce its instruction cache power by 78%.

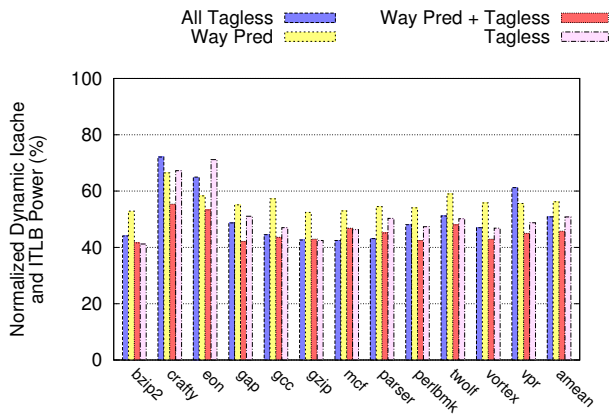
For *crafty*, 40% of the accesses are in tagged mode. This is because the linker could not place a large fraction of their code within tagless regions without incurring penalties from switching between tagless regions. In these cases, it is better to keep the code in tagged regions to avoid invalidating tagless regions too often when this switching occurs (see section III-C). As figure 7 shows, in these applications about half the banks as normal are accessed. This is confirmed by figure 8b which shows that *crafty* cannot often access the cache in tagless mode. However, this benchmark still achieves good instruction cache power savings of around 30%.

### B. Comparison With Other Approaches

This section compares our technique to alternative hardware and software schemes, shown in figure 9. The first technique is labeled *All Tagless* and represents a binary with all code placed into tagless program regions, then run on a cache using our tagless hardware. The second approach (*Way Pred*) show the baseline binaries run on a cache with way prediction support by Powell et al. [23]. In addition, we show a combination of our tagless scheme and way prediction (*Way Pred + Tagless*) which is evaluated using the same tagless binaries as our technique. Finally, for reference, we show our approach as the final bar, labeled *Tagless* again. This has already been analyzed in section V-A.




(a) Performance




(b) Icache and ITLB dynamic power

Fig. 9. Comparing caching schemes. We show a binary where all code is placed in tagless regions (*All Tagless*), a cache with way prediction (*Way Pred*), way prediction and our tagless scheme (*Way Pred + Tagless*), and our approach alone (*Tagless*).

*f) Performance:* As can be seen in figure 9a, laying out the entire binary for tagless accessing causes large overall slowdown. This is due to conflicts between the tagless regions within the instruction cache requiring frequent invalidations of entire regions. The performance characteristics of the other schemes are similar to those of running on tagless hardware, as described in section V-A. In some cases, *Way Pred* and *Way Pred + Tagless* perform slightly worse due to the inaccuracies in the way prediction hardware. This causes a maximum of 4% performance loss (for *crafty*).

*g) Power:* Figure 9b shows that the *All Tagless* technique can be detrimental to power due to the many tagless region conflicts within the instruction cache. Tagless caching with way prediction (*Way Pred + Tagless*) achieves a 10% improvement over regular way prediction, 5% better than our regular *Tagless* scheme. This demonstrates that our scheme is in part orthogonal to the savings achieved by way prediction, and that both can be combined for improved efficiency. The marginal power savings gained by way prediction are, however, fairly small, and using the simpler *Tagless* hardware platform may be preferable.

### C. Impact On SMT

Now that we have demonstrated the impact of our approach against other power saving schemes, this section performs an evaluation of our technique on an SMT processor, where the instruction cache is shared between two independent workloads. This situation enables us to evaluate our tagless caching scheme in an environment where there is significant contention within the cache as the workloads compete for cache resources. Figure 10 shows the results for each randomly-selected pair of benchmarks.

*h) Performance:* Figure 10a shows that the hardware-only element or our approach again has a significant impact in terms of performance on an SMT processor. On the other hand, the software element and our hybrid scheme have a negligible impact with only *crafty.gap* having significant slowdown due to the contention in the instruction cache. On average there is no change in performance across our benchmark pairings for our hybrid scheme.

*i) Power:* As figure 10b shows, significant power savings can still be achieved on an SMT processor using tagless instruction caching. On average, our tagless scheme requires only 59% of the I-cache and ITLB cache power compared with the baseline. The best scheme (*twolf.bzip2*) requires less than 50% of the baseline's power. These results emphasize the ability of our approach to enable significant power savings, even within an environment with high contention for resources.

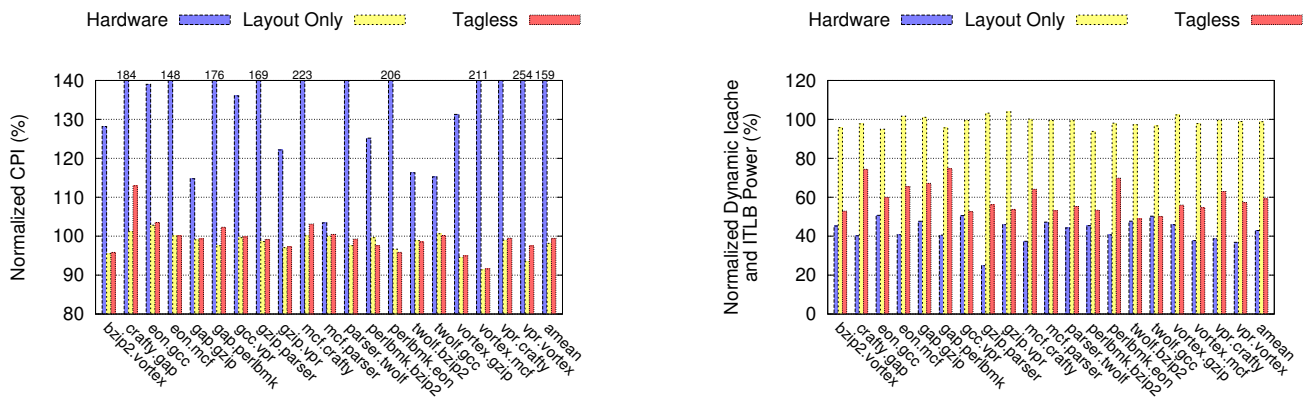
### D. Summary

This section has presented the results of our tagless instruction caching scheme. Using our approach on a superscalar processor provides dynamic power savings of 66% in the instruction cache on average when using a 32kB cache, 49% considering both I-cache and the ITLB with no performance impact.

## VI. Related Work

There is a rich body of related work that aims at reducing cache power consumption while maintaining high performance. Ravindran et al. [24] employed a scratchpad memory (SPM) alongside the instruction cache which was filled with the most frequently-executed basic blocks, found through profiling on the compiler's intermediate representation. As in Verma et al. [28], specific instructions are used to load and unload SPM content. Chen et al. [6] used a completely software managed SPM without an instruction cache at all. Dally et al. [7] replaced the cache with explicitly managed registers, leaving the management of this instruction register file to the compiler. However, these schemes require additional instructions to be included in the ISA and are therefore not backwards compatible.

Egger et al. [10] presented a similar scheme, but avoided the ISA extensions by filling the SPM using a run time memory manager combined with MMU-based page protection.




(a) Performance

(b) Icache and ITLB dynamic power

Fig. 10. Results of our tagless accessing approach along with the hardware and software elements on a dual-threaded SMT processor.

Furthermore, they replace the cache with a simple direct-mapped "minicache" but this causes performance losses when unmodified binaries are run. Zhang et al. [35] used compiler analysis to trigger a low-power leakage mode for instruction cache lines that were not likely to be used for a long time. However, this scheme targets static power reduction and is therefore orthogonal to the work we present in this paper.

Yang et al. [33] proposed a compiler-assisted technique for inducing selective Dcache *pinning*, so as to avoid replacement of heavily referenced data in an XScale processor. Zhu and Tay [37] proposed a compiler restructuring technique that both reduces the working-set size and inserts special cache-scaling instructions into the code to turn off unused cache lines. In Bellas et al. [3] an additional mini-cache for critical loop code is presented, along with a compiler strategy for its efficient usage. Zhuang et al. [38] took a slightly different perspective and proposed a compiler-directed code and data restructuring algorithm aimed at improving the power efficiency of cache prefetching.

Within hardware, there has also been significant interest in power-efficient caching. A technique that can guarantee hits was proposed by Hines et al. [13], completely avoiding tag checks for the majority of instructions, similar to way-memoization, proposed by Ma et al. [21]. Way prediction has also been used to reduce the number of tags and data ways accessed within a set-associative cache. Calder and Grunwald [5] added a tagless prediction table that points directly into the instruction cache. This allowed them to fetch instructions following a branch from a single set and way while the target address was being calculated. Powell et al. [23] built on this work by providing way prediction for all instruction fetches, rather than solely after branches. We compare against the latter scheme and show how our approach saves more power with less performance loss.

Other researchers have proposed heterogeneous way sizes [1], disabling ways [2], and using near threshold tolerant ways [9] to fit the cache to the varying application requirements. Furthermore, there has been significant work in designing modified cache hierarchies with the aim of saving power [17], [19], [32]. In the data cache, researchers have provided compiler hints about lines to keep and evict [25], [30].

There has also been work that targets the data cache for thermal optimization [14], [18], to improve power consumption or performance by using a scratchpad memory [22], [26], and to take advantage of frequently-occurring values [29], [34]. Finally, there are many hardware schemes that seek to reduce a cache's leakage energy by placing cache lines in state-preserving and state-destroying low power sleep modes [11], [15], [16], [35], [36].

## VII. Conclusions and Future work

This paper has presented a novel cache architecture and linker algorithm to provide tagless instruction caching for power saving. We use the linker to place frequently-executed code in specific regions within the program binary that are then accessed from the cache without tag checks. Our approach is extremely versatile, having been evaluated on a super-scalar processor with and without multi-threading support. Furthermore, it requires no ISA modifications at all, allowing backwards compatibility with existing, unmodified binaries.

## Acknowledgements

The authors would like to thank Koen De Bosschere and Christian Fensch for comments and feedback on their draft manuscript. This work was supported by the UK's Royal Academy of Engineering and EPSRC, and was partially supported by the IT FIRB PHOTONICA (RBFR08LE6V) and IT PRIN 2008 (200855LRP2) projects. It has made use of the Stevin Supercomputer Infrastructure at Ghent University and the Edinburgh Compute and Data Facility (ECDF) (http://www.ecdf.ed.ac.uk/). The ECDF is partially supported by the eDIKT initiative (http://www.edikt.org.uk). Jones, Bartolini and Maebe are members of the European Network of Excellence on High Performance and Embedded Architecture and Compilation (HiPEAC).

## REFERENCES


[1] J. Abella and A. González, "Heterogeneous way-size cache," in *ICS - International Conference on Supercomputing*, 2006.

[2] D. H. Albonesi, "Selective cache ways: on-demand cache resource allocation," in *MICRO - International Symposium on Microarchitecture*, 1999.

[3] N. Bellas, I. Hajj, C. Polychronopoulos, and G. Stamoulis, "Architectural and compiler techniques for energy reduction in high-performance microprocessors," *IEEE Trans. on VLSI Systems*, vol. 8, no. 3, 2000.

[4] N. L. Binkert, R. G. Dreslinski, L. R. Hsu, K. T. Lim, A. G. Saidi, and S. K. Reinhardt, "The M5 simulator: Modeling networked systems," *IEEE Micro*, vol. 26, 2006.

[5] B. Calder and D. Grunwald, "Next cache line and set prediction," in *ISCA - International Symposium on Computer Architecture*, 1995.

[6] G. Chen, I. Kayadif, W. Zhang, M. Kandemir, I. Kolcu, and U. Sezer, "Compiler-directed management of instruction accesses," in *Euromicro Symposium on Digital System Design*, 2003.

[7] W. J. Dally, J. Balfour, D. Black-Shaffer, J. Chen, R. C. Harting, V. Parikh, J. Park, and D. Sheffield, "Efficient embedded computing," *Computer*, vol. 41, no. 7, 2008.

[8] B. De Sutter, B. De Bus, and K. De Bosschere, "Link-time binary rewriting techniques for program compaction," *ACM TOPLAS - Trans. on Programming Languages and Systems*, vol. 27, no. 5, 2005.

[9] R. Dreslinski, G. Chen, T. Mudge, D. Blaauw, D. Sylvester, and K. Flautner, "Reconfigurable energy efficient near threshold cache architectures," in *MICRO - International Symposium on Microarchitecture*, 2008.

[10] B. Egger, J. Lee, and H. Shin, "Dynamic scratchpad memory management for code in portable systems with an mmu," *ACM TECS - Trans. in Embedded Computing Systems*, vol. 7, no. 2, 2008.

[11] K. Flautner, N. S. Kim, S. Martin, D. Blaauw, and T. Mudge, "Drowsy caches: Simple techniques for reducing leakage power," in *ISCA - International Symposium on Computer Architecture*, 2002.

[12] A. Hasegawa, I. Kawasaki, K. Yamada, S. Yoshioka, S. Kawasaki, and P. Biswas, "Sh3: High code density, low power," *IEEE Micro*, vol. 15, no. 6, 1995.

[13] S. Hines, D. Whalley, and G. Tyson, "Guaranteeing hits to improve the efficiency of a small instruction cache," in *MICRO - International Symposium on Microarchitecture*, 2007.

[14] J. K. John, J. S. Hu, and S. G. Ziavras, "Optimizing the thermal behavior of subarrayed data caches," in *ICCD - International Conference on Computer Design*, 2005.

[15] S. Kaxiras, Z. Hu, and M. Martonosi, "Cache decay: Exploiting generational behavior to reduce cache leakage power," in *ISCA - International Symposium on Computer Architecture*, 2001.

[16] N. S. Kim, K. Flautner, D. Blaauw, and T. Mudge, "Drowsy instruction caches: Leakage power reduction using dynamic voltage scaling and cache sub-bank prediction," in *MICRO - International Symposium on Microarchitecture*, 2002.

[17] J. Kin, M. Gupta, and W. Mangione-Smith, "The filter cache: an energy efficient memory structure," in *MICRO - International Symposium on Microarchitecture*, 1997.

[18] J. C. Ku, S. Ozdemir, G. Memik, and Y. Ismail, "Thermal management of on-chip caches through power density minimization," in *MICRO - International Symposium on Microarchitecture*, 2005.

[19] L. H. Lee, B. Moyer, and J. Arends, "Instruction fetch energy reduction using loop caches for embedded applications with small tight loops," in *ISLPED - International Symposium on Low Power Electronics and Design*, 1999.

[20] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi, "McPAT: An integrated power, area, and timing modeling framework for multicore and manycore architectures," in *MICRO - International Symposium on Microarchitecture*, 2009.

[21] A. Ma, M. Zhang, and K. Asanović, "Way memoization to reduce fetch energy in instruction caches," in *WCED - Workshop on Complexity-Effective Design (ISCA)*, 2001.

[22] N. Nguyen, A. Dominguez, and R. Barua, "Memory allocation for embedded systems with a compile-time-unknown scratch-pad size," in *CASES - Int. Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, 2005.

[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 *MICRO - International Symposium on Microarchitecture*, 2001.

[24] R. Ravindran, P. Nagarkar, G. Dasika, E. Marsman, R. Senger, S. Mahlke, and R. Brown, "Compiler managed dynamic instruction placement in a low-power code cache," in *CGO - International Symposium on Code Generation and Optimization*, 2005.

[25] J. B. Sartor, S. Venkiteswaran, K. S. McKinley, and Z. Wang, "Cooperative caching with keep-me and evict-me," in *INTERACT - Annual Workshop on Interaction between Compilers and Computer Architectures*, 2005.

[26] A. Shrivastava, I. Issenin, and N. Dutt, "Compilation techniques for energy reduction in horizontally partitioned cache architectures," in *CASES - Int. Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, 2005.

[27] "The Standard Performance Evaluation Corporation (SPEC) CPU 2000 Benchmark Suite."

[28] M. Verma, L. Wehmeyer, and P. Marweclel, "Dynamic overlay of scratchpad memory for energy minimization," in *CODES + ISSS - International conference on Hardware/software codesign and system synthesis*, 2004.

[29] L. Villa, M. Zhang, and K. Asanović, "Dynamic zero compression for cache energy reduction," in *MICRO - International Symposium on Microarchitecture*, 2000.

[30] Z. Wang, K. S. McKinley, A. L. Rosenberg, and C. C. Weems, "Using the compiler to improve cache replacement decisions," in *PACT - Int. Conference on Parallel Architectures and Compilation Techniques*, 2002.

[31] R. E. Wunderlich, T. F. Wenisch, B. Falsafi, and J. C. Hoe, "SMARTS: Accelerating microarchitecture simulation via rigorous statistical sampling," in *ISCA - International Symposium on Computer Architecture*, 2003.

[32] C.-L. Yang and C.-H. Lee, "Hotspot cache: joint temporal and spatial locality exploitation for i-cache energy reduction," in *ISLPED - International Symposium on Low Power Electronics and Design*, 2004.

[33] H. Yang, R. Govindarajan, G. R. Gao, and Z. Hu, "Improving power efficiency with compiler-assisted cache replacement," *Journal of Embedded Computing*, vol. 1, no. 4, 2005.

[34] J. Yang and R. Gupta, "Energy efficient frequent value data cache design," in *MICRO - International Symposium on Microarchitecture*, 2002.

[35] W. Zhang, J. Hu, V. Degalahal, M. Kandemir, N. Vijaykrishnan, and M. Irwin, "Compiler-directed instruction cache leakage optimization," in *MICRO - International Symposium on Microarchitecture*, 2002.

[36] W. Zhang, M. Karakoy, M. Kandemir, and G. Chen, "A compiler approach for reducing data cache energy," in *ICS - International Conference on Supercomputing*, 2003.

[37] X. Zhu and T. Tay, "A compiler-controlled instruction cache architecture for an embedded low power microprocessor," in *International Conference Computer and Information Technology*, 2005.

[38] X. Zhuang and S. Pande, "Power-efficient prefetching for embedded processors," *ACM TECS - Trans. in Embedded Computing Systems*, vol. 6, no. 1, 2007.