# Compiler Managed Dynamic Instruction Placement in a Low-Power Code Cache

Rajiv A. Ravindran[1], Pracheeti D. Nagarkar[1], Ganesh S. Dasika[1],
Eric D. Marsman[1], Robert M. Senger[1], Scott A. Mahlke[1], and Richard B. Brown[2]

[1]Dept. of EECS, University of Michigan, Ann Arbor
[1]{rravindr, pnagarka, gdasika, emarsman, rsenger, mahlke}@umich.edu
[2]College of Engineering, University of Utah, Salt Lake City
[2]brown@coe.utah.edu

## ABSTRACT

Modern embedded microprocessors use low power on-chip memories called scratch-pad memories to store frequently executed instructions and data. Unlike traditional caches, scratch-pad memories lack the complex tag checking and comparison logic, thereby proving to be efficient in area and power. In this work, we focus on exploiting scratch-pad memories for storing hot code segments within an application. Static placement techniques focus on placing the most frequently executed portions of programs into the scratch-pad. However, static schemes are inherently limited by not allowing the contents of the scratch-pad memory to change at run time. In a large fraction of applications, the instruction memory footprints exceed the scratch-pad memory size, thereby limiting the usefulness of the scratch-pad. We propose a compiler managed dynamic placement algorithm, wherein multiple hot code sequences, or traces, are overlapped with each other in the scratch-pad memory at different points in time during execution. Special copy instructions are provided to copy the traces into the scratch-pad memory at run-time. Using a power estimate, the compiler initially selects the most frequent traces in an application for relocation into the scratch-pad memory. Through iterative code motion and redundancy elimination, copy instructions are inserted in infrequently executed regions of the code. For a 64-byte code cache, the compiler managed dynamic placement achieves an average of 64% energy improvement over the static solution in a low-power embedded microcontroller.

## 1. INTRODUCTION

With the proliferation of cellular handsets, digital cameras, and other portable computing systems, power consumption in microprocessors has become a dominant design concern. Power consumption directly affects both battery lifetime and the amount of heat that must be dissipated, thus it is critical to create power-efficient designs. However, many of these devices perform computationally demanding processing of images, sound, video, or packet streams. Thus, simply scaling voltage and frequency to reduce power is insufficient as the desired performance level cannot be achieved. Hardware and software solutions that maintain performance while reducing power consumption are required.

With embedded processors, the instruction fetching subsystem can contribute to a large fraction of the total power dissipated by the processor. For example, instruction fetch contributes 27% in the StrongARM SA-110 [11], and almost 50% for the Motorola MCORE [19] of the total processor power. Intuitively, this makes sense as instruction fetch is one of the most active portions of a processor. Instructions are fetched nearly every cycle, involving one or more memory accesses, some of which may be off-chip accesses. For this paper, we focus on reducing instruction fetch power.

A number of approaches have been taken by designers to reduce instruction fetch energy. First, more efficient instruction cache designs can be employed to reduce dynamic or leakage power [16]. Second, instruction compression techniques can be employed to reduce the number of instruction bits that need to be fetched [21]. Or third, bus encoding schemes can be employed to reduce the number of bits that switch each cycle [6].

Another approach that is particularly effective for embedded systems is to use loop caches (LCs) [17, 5, 20, 10, 30, 29]. LCs are small instruction buffers that can be designed to have extremely low-power per access. They are most effective when execution is dominated by small loops whose bodies can reside entirely within the LC. LCs can be broadly classified into two categories: hardware or software managed.

With a hardware managed approach, loops are dynamically copied into the LC and fetch is re-directed from the L1 instruction cache to the LC using limited hardware support [10, 20]. Hardware managed caches, referred to as filter or L0 caches [17], use cache-like tags but are often small and direct-mapped, hence their power characteristics and access time are much better than those of conventional caches. But, they suffer from high miss rates and cache management overhead.

To eliminate the overhead of tag comparisons, a tag-less LC has been proposed [20, 19]. Here, the LC is a small instruction buffer placed between the processor and the L1 instruction cache. A LC controller is responsible for identifying recurring code segments in the dynamic instruction stream, filling the LC, and redirecting fetch to the LC. This design can be more power efficient than the hardware tagged approach. However, there are several negatives of this approach, including LC controller complexity, the inability to relocate loops with control flow or subroutine calls, and the controller may make poor choices as it does not have a complete view of the program execution.

Conversely, software managed LCs (also called code caches or scratch-pad memories [22, 28]) reduce hardware management overhead by relying on the compiler to insert code segments into the LC. A recent study [4] showed that scratch-pad memory has 40% lower power consumption than a cache of equivalent size. The most common strategy is to statically map hot blocks into the LC using profile information [5, 15, 24]. Software static schemes have the advantage of no run-time copy overhead. Further, a global optimal placement can be performed to maximize the LC effectiveness over the entire run of the application. However, the major negative is that the LC contents

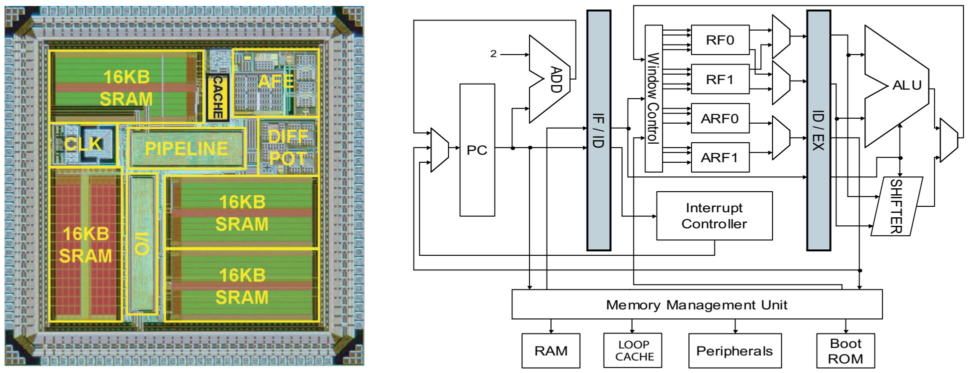


**Figure 1: The WIMS microcontroller in TSMC 0.18$\mu$m CMOS and the WIMS pipeline with the loop cache.**

cannot change during execution.

Software static schemes break down when a program has multiple important loops that collectively cannot fit in the LC. As a result, only a subset of the loops can be mapped into the LC. To overcome this problem, compiler-directed dynamic placement has been recently proposed [25, 31]. With this approach, the compiler inserts copy instructions into the program to copy blocks of instructions into the LC and redirect instruction fetch to the LC. As a result, the compiler can change the contents of the LC during program execution as it desires by inserting copy instructions at the appropriate locations. Compiler-directed dynamic placement has the potential to combine the benefits of the hardware-based schemes with the low-overhead of the software-based schemes. Previous approaches to dynamic placement use an integer linear programming (ILP) technique to find an optimal placement of instructions/data into the LC [25, 31]. But ILP-based approaches may not be practical in terms of run-time and often fail for moderate to large sized applications.

In this work, we propose a new approach for compiler-directed dynamic placement. An inter-procedural heuristic for identifying hot instruction traces to insert in the LC is proposed. Based on a profile-driven power estimate, the selected traces are then packed into the LC by the compiler, possibly sharing the same space, such that the run-time cost due to copying the traces, is minimized. Through iterative code motion and redundancy elimination, copy instructions are inserted in infrequently executed regions of the code to copy traces into the LC. The approach works with arbitrary control flow and is capable of inserting any code segment into the LC (i.e., not just a loop body). A more detailed comparison of our work with the ILP-based solution is provided in Section 4.3.

## 2. WIMS ARCHITECTURE

### 2.1 Overview

Our work in this paper is based on a real hardware platform that was designed to control a variety of miniature, low-power embedded sensor systems called the WIMS (Wireless Integrated Microsystems) microcontroller [23]. The microcontroller fabricated in TSMC 0.18$\mu$m CMOS is shown in Figure 1 and consists of three major sub-blocks: the digital core, the analog front-end, and the CMOS-MEMS clock reference. Power minimization was a key design constraint for each of the sub-blocks.

A 16-bit load/store architecture with dual-operand register-to-register instructions was chosen to satisfy the power and performance requirements of the microcontroller. A 16-bit datapath was selected to reduce the complexity and power consumption of the core while providing adequate precision in calculations, given that the sensors controlled by this chip require 12 bits of resolution. A unified 24-bit address space for data and instruction memory satisfies the potentially large storage requirements of remote sensor systems. The current implementation of the core has four 16Kb banks of on-chip SRAM with a memory management unit that disables inactive banks.

A 16-bit WIMS instruction set was custom designed and includes 77 instructions and eight addressing modes. The 16-bit instruction encoding was chosen so as to reduce the instruction memory footprint and thus the instruction fetch energy. The core contains sixteen 16-bit data registers that are split into two register windows with eight data registers each. Similarly, four 24-bit address registers are evenly split into two register windows.

**WIMS Loop Cache:** The WIMS design is somewhat atypical of most processors used in previous research in that it contains no caches. Caches were not needed because memory accesses to the on-chip SRAM banks complete in one cycle. Moreover, the area/power overhead associated with the tag memory and logic for tag comparisons of conventional cache organizations did not make sense in the design. However, instruction fetch contributes a large fraction of the overall power dissipation of the chip (around 30% for the WIMS processor), thus a simple, software-managed code cache was added to the design. Note that we shall refer to the code cache as an LC for consistency with previous papers though there is no limitation of storing only loop bodies in it. The LC is a small SRAM (512-bytes in the current design) that can be designed with substantially lower power dissipation characteristics than the 16KB banks used for the rest of the memory ("main memory"). Figure 1 shows the architecture block diagram of the WIMS microcontroller with the LC. The LC occupies a range of the physical address space, thus copying instructions into the LC corresponds to copying instructions into those specific physical addresses.

The goal of the compiler support proposed in this paper is to make effective use of the LC by dynamically copying instructions into it for programs with general control structures. To this end, a single instruction was added to the WIMS architecture, *LC_COPY*. The *LC_COPY* takes three source operands: the address of the first instruction of the region of code to be copied (PC relative), the starting chunk in the LC to begin placement, and the number of chunks to copy. The LC is logically divided into chunks, each being a fixed size (16 bytes for our experiments). A chunk represents minimum granularity at which copies can occur. By subdividing the LC into chunks, fewer bits to encode the operands were needed which was important to fit into the 16-bit encoding. The *LC_COPY* instruction copies *number of chunks * size per chunk* bytes into the LC beginning at the starting chunk. The processor stalls while the copy takes place. The copying can be implemented using a direct memory access

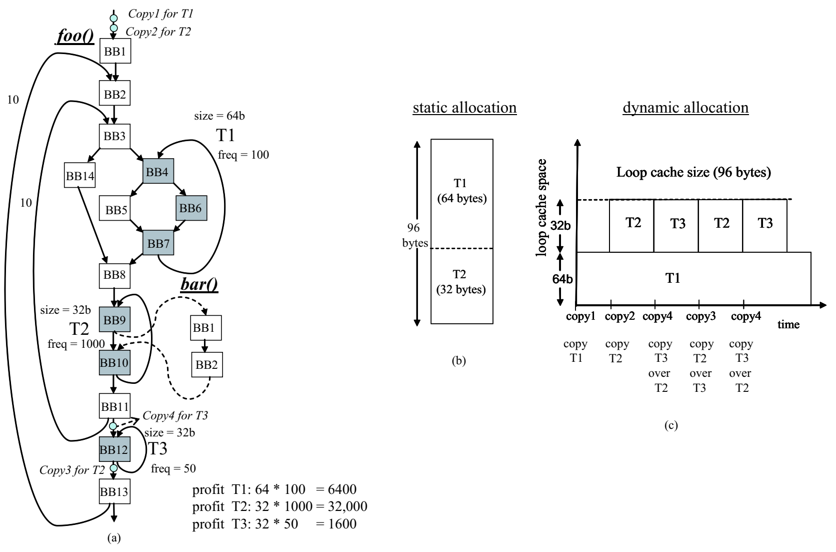


**Figure 2: Example (a) weighted control flow graph (b) static allocation (c) dynamic allocation as a function of time.**

engine or as a software interrupt. In our experimental studies, we assume that the *LC_COPY* can copy 2-bytes per cycle.

Targets of branches into regions of the code that have been selected by the compiler are modified to point to the addresses in the LC where the region would be placed dynamically by the WIMS assembler/linker. Instruction fetch is thus redirected to the LC whenever control enters into a selected code region.

## 2.2 Dynamic Placement Motivation

To demonstrate the issues and benefits of dynamically copying instructions into the LC, consider the example shown in Figure 2. Figure 2(a) shows a control flow graph consisting of three hot regions shaded in gray. The shaded regions represent frequently executed sequence of basic blocks (BBs) in the code called *traces* [13]. Trace T1 consists of BBs 4, 6, and 7, T2 of BBs 9 and 10, while T3 contains a single BB, 12. These traces were identified by profiling the program on a sample input. Traces can include either a whole loop (e.g., T3), a part of a loop (e.g., T1), embedded procedure calls (e.g., T2), or any other complex control flow. The traces are annotated with the profile weights (frequency) and size in bytes.

For illustration, assume the LC size is 96-bytes. The trace profit, which measures the desirability of placing a trace in the LC, is given by its size in bytes times the profile weight (Figure 2(a)). Figure 2(b) shows the contents of the LC for a static allocation scheme. As the LC can hold only 96-bytes, the static scheme packs only the top two profitable traces, T1 and T2, of sizes 64 and 32-bytes, respectively, into the LC. But, the dynamic scheme (Figure 2(c)), is able to allocate all traces by inserting copy instructions as shown on the edges in Figure 2(a). Copy 1 (for T1) is executed once before entering the inner loop, thus T1 remains in the LC throughout its lifetime. Copies 2 and 3 (both for T2), and copy 4 (for T3) alternately insert T2 and T3 into the same location in the LC. Each copy ensures that the trace is inserted into the LC before they are executed. It should be noted that copy 3 and copy 4 have to be placed within the outer loop to copy traces T2 and T3 prior to their execution. By effectively overlapping multiple blocks of code and placing copies appropriately, the dynamic scheme is able to capture all the hot regions and thus achieve better LC utilization. This approach was first used in pre-virtual memory management operating systems for overlaying code for different processes.

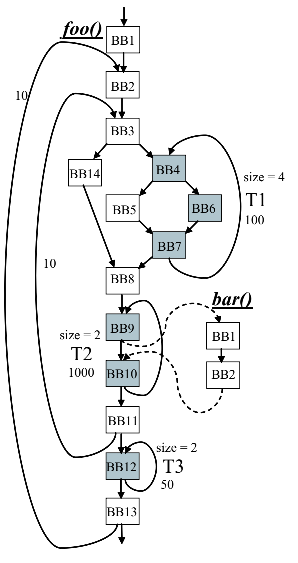
## 3. DYNAMIC PLACEMENT

### 3.1 Overview

The dynamic instruction placement scheme has been implemented within the Trimaran [27] compiler framework. The compiler frontend performs control flow profiling and annotates the intermediate representation (IR) with *traces* [13]. Traces are frequently executed linear sequences of basic blocks that are contiguously laid out in memory [9]. Traces are formed with a 60% probability of an in-trace transition and with size limited to that of the LC. These traces are considered as candidates for placement into the LC. The dynamic placement phase, based on the execution profile information, then inserts the *LC_COPY* instructions into the IR. The WIMS assembler/linker assigns instructions to physical memory locations including adjusting of the branch targets for the relocated code.

The dynamic placement algorithm has two objectives: (i) select traces from the program and place them into locations in the LC such that the energy benefit is maximized, and (ii) place copy instructions so that traces are copied prior to execution while minimizing the overhead due to copying. To achieve these objectives, the dynamic placement is divided into two distinct phases - trace selection/placement and copy placement. The trace selection/placement phase, using the execution profile information and the annotated traces from the IR, selects the most beneficial traces and decides where they are to be placed in the LC using an energy benefit heuristic. The placement phase could possibly overlap the traces within the LC. The placement decisions are driven by energy considerations and do not take performance into account. Following this, the copy placement phase naïvely inserts copy operations on every entry edge of a selected trace in the IR. This ensures that whenever control reaches a trace that has been placed in the LC, it is copied prior to execution. Many of these copies may be redundant or present on highly executed paths, thus causing high copy overhead. Based on a liveness analysis scheme, the copies are then hoisted in the control flow graph (CFG) across procedure boundaries to less frequently executed blocks so as to reduce the copy overhead while maintaining correctness of execution.

The example in Figure 2 is used throughout this section to illustrate how the candidate traces, T1, T2, and T3 are placed in the LC and how subsequent copy insertion and hoisting are



(a)

Benefit:

ProfileWeight * (TraceSize * ChunkSize)/FetchSize * (MMFetchEnergy - LCFetchEnergy)

T1: 100*((4*16)/2 )*(2-1)  = 3200 nJ
T2: 1000*((2*16)/2)*(2-1) = 16,000 nJ
T3: 50*((2*16)/2)*(2-1)     = 800 nJ

CopyCost:

(TraceSize * ChunkSize)/FetchSize * (MMFetchEnergy + LCWriteEnergy)

T1: ((4*16)/2)*(2+1) = 96 nJ
T2: ((2*16)/2)*(2+1) = 48 nJ
T3: ((2*16)/2)*(2+1) = 48 nJ

(b)

NetGain = Benefit – CopyCost

T1: 3200 – 96   = 3104 nJ
T2: 16000 – 48 = 15952 nJ
T3: 800 – 48     = 752 nJ

(c)

**Figure 3: Trace selection and placement example. (a) CFG (b) Benefit and CopyCost computation for traces T1, T2, and T3.**

performed for these selected traces. The CFG is redrawn in Figure 3(a) for convenience. The trace selection/placement and the copy placement phases are detailed in the sections below.

## 3.2 Trace Selection/Placement

The trace selection phase takes as input the IR annotated with the traces. Traces are chosen as candidates for LC allocation for two reasons. First, they help reduce the number of copies as a single copy instruction can copy a large amount of frequently executed code, like a loop body, into the LC. Second, a trace is a high frequency path of execution consisting of basic blocks connected by fall-through edges; thus, the number of control flow transfers in and out of the LC is reduced. Conversely, traces are fine grained enough to enable selection of small hot program segments for general applications.

Trace selection/placement involves picking traces and placing them in the LC such that there is a savings in instruction fetch energy. If a trace is placed in the LC, then whenever the trace is executed, it has to be executed out of the LC. This is required as all branches into the trace have their offsets changed to the location in the LC where the trace will be placed. Thus, the trace needs to be copied prior to execution which involves a copy overhead. Also, the exact placement of the trace in the LC is important because if traces overlap in the LC, repeated copies may be required. The trace selection/placement algorithm selects and places a trace at a particular location in the LC only if there is an overall energy benefit for that trace. The trace selection/placement consists of two steps - (i) computing the energy gain for every trace, and (ii) placing the trace into the LC.

**Computing Trace Energy Gain:** For a trace to be considered as a candidate for placement in the LC, the energy savings obtained in executing the trace out of the LC must be greater than a one-time copy overhead. Thus, traces with a higher copy overhead than the potential energy gain are non-beneficial and can be filtered out. For every trace, $T_i$, the copy cost and benefit of placing the trace in the LC is initially computed assuming that the LC is of infinite size and the trace does not overlap with any other trace. Since copying into the LC takes place at the chunk granularity, the size of the trace is computed in number of chunks. In Figure 3(a), T1, T2, and T3 are assumed to take 4, 2, and 2 chunks, respectively. The cost of copying a trace into the LC is the sum of the energy needed to fetch the trace from the main memory and write the trace into the LC. The copy cost (measured in nJoules) is given by the equation:

$$\mathit{CopyCost}(T_i) = \frac{\mathit{TraceSize}(T_i) * \mathit{ChunkSize}}{\mathit{FetchSize}} * (\mathit{MMFetchEnergy} + \mathit{LCWriteEnergy}) \quad (1)$$

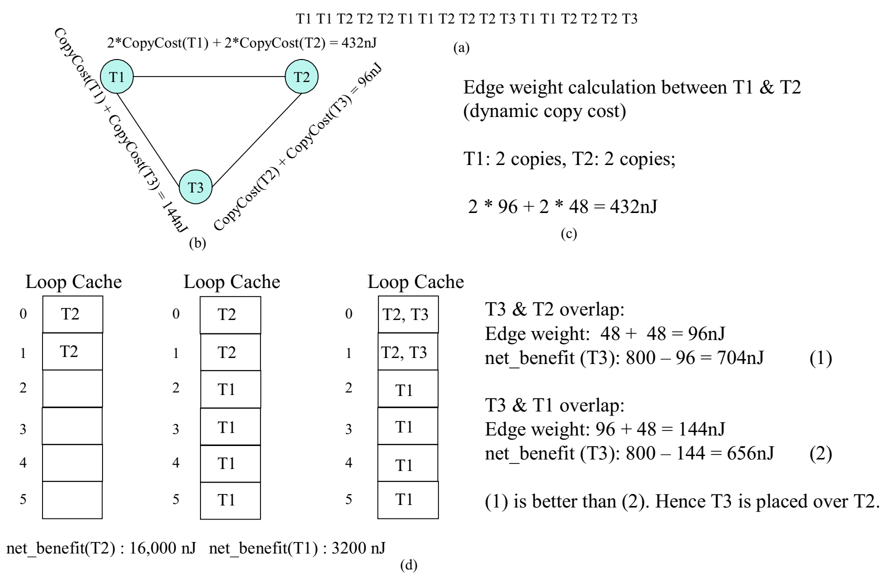
where $TraceSize(T_i)$ is the size of $T_i$ in chunks, $ChunkSize$ is the size of a single chunk (assumed 16-bytes), and $FetchSize$ is the number of bytes accessed per fetch from main memory to the LC (assumed 2-bytes per access). $MMFetchEnergy$ and $LCWriteEnergy$ are the energy required for a single fetch from main memory and a single write into the LC, respectively.

The benefit of placing a trace in the LC is the savings in energy obtained when the trace is executed out of the LC as opposed to executing from main memory. The benefit (measured in nJoules) is given by the equation:

$$\mathit{Benefit}(T_i) = \mathit{ProfileWeight}(T_i) * \frac{\mathit{TraceSize}T_i * \mathit{ChunkSize}}{\mathit{FetchSize}} * (\mathit{MMFetchEnergy} - \mathit{LCFetchEnergy}) \quad (2)$$

where $LCFetchEnergy$ is the energy required for a single fetch out of the LC and $ProfileWeight(T_i)$ is the execution frequency of $T_i$ obtained through profiling. The calculation of $CopyCost$ and $Benefit$ for traces T1, T2, and T3 for the running example are shown in Figure 3(b). Here, we assume that the main memory fetch energy is 2 nJ, while LC fetch/write energy is 1 nJ. The net energy gain of placing a trace is the difference between the benefit and the copy cost defined as, $NetGain(T_i) = Benefit(T_i) - CopyCost(T_i)$, as shown in Figure 3(c). Traces for which the net gain is less than zero are not considered further for placement.

**Placing Traces into the Loop Cache:** The placement algorithm decides where each trace is placed in the LC. A trace occupies continuous locations in memory and hence is assigned to a sequence of contiguous chunks. Thus, the placement algorithm has to decide on the best starting chunk. If a given trace solely occupies a sequence of chunks, then the benefit of placing the trace is the same as $NetGain$. But, this is simply static placement. Dynamic placement allows multiple traces to occupy the same LC chunk. However, a trace must be recopied whenever it



T1 T1 T2 T2 T2 T1 T1 T2 T2 T2 T3 T1 T1 T2 T2 T2 T3
(a)
2*CopyCost(T1) + 2*CopyCost(T2) = 432nJ
CopyCost(T1) + CopyCost(T3) = 144nJ
CopyCost(T2) + CopyCost(T3) = 96nJ
T1
T2
T3
(b)
Edge weight calculation between T1 & T2
(dynamic copy cost)
T1: 2 copies, T2: 2 copies;
2 * 96 + 2 * 48 = 432nJ
(c)
Loop Cache
T2, T3
T3 & T2 overlap:
Edge weight: 48 + 48 = 96nJ
net_benefit (T3): 800 – 96 = 704nJ (1)
T3 & T1 overlap:
Edge weight: 96 + 48 = 144nJ
net_benefit (T3): 800 – 144 = 656nJ (2)
(1) is better than (2). Hence T3 is placed over T2.
net_benefit(T2) : 16,000 nJ net_benefit(T1) : 3200 nJ
(d)

**Figure 4:** **Trace selection and placement example. (a) Dynamic execution trace (b) Temporal relationship graph. (c) Edge weight calculation between nodes T1 and T2 (d) Placement of T1, T2, and T3 into the LC.**

gets displaced by an overlapping trace. Dynamic placement is successful when the combined benefit of placing multiple traces is more than the overhead due to repeated copying.

In order to compute the overhead due to repeated copying, the previous cost/benefit analysis is extended to account for the additional copying cost (*Dynamic_Copy_Cost*). To this end, a temporal relationship graph [14] (TRG) is constructed based on a *dynamic execution trace*. The dynamic execution trace consists of traces and is obtained during execution profiling. Figure 4(a) shows the dynamic execution trace for a sample run of the program in Figure 3(a). The TRG helps to estimate the number of dynamic recopies required if two traces overlap in the LC.

The nodes in the TRG are the traces, while the edges are annotated with the dynamic copy cost. Between every pair of nodes $T_i$ and $T_j$, the edge weight denotes the number of copies of $T_i$ ($CopyCost(T_i)$) for any $T_j$ that occurs between every two consecutive occurrences of $T_i$ in the dynamic execution trace. A $T_j$ occurring between two consecutive instances of $T_i$ implies that $T_i$ needs to be recopied prior to its second occurrence, if $T_i$ and $T_j$ overlap in the LC. The TRG is constructed by linearly scanning the input dynamic execution trace and maintaining a queue of currently seen traces. Each new trace $T_i$, seen in the input, is added to the queue. The queue is then scanned, starting from the tail, for a previous occurrence of $T_i$. For every unique trace $T_j$ seen prior to the previous occurrence of $T_i$, the edge weight between $T_i$ and $T_j$ is incremented by the copy cost of $T_i$. The previous occurrence of $T_i$ is then deleted from the queue as the new instance of $T_i$ becomes the next previous occurrence.

The TRG for the dynamic execution trace in Figure 4(a) is shown in Figure 4(b). Considering T1 and T2 alone, there is an instance of T2 between every instance of T1 and vice-versa. Thus, if T1 and T2 overlap in the LC, two recopies of both T1 and T2 are required. The edge weight between nodes T1 and T2 is therefore $2 * CopyCost(T1) + 2 * CopyCost(T2)$ (Figure 4(c)), where $CopyCost$ is computed as shown earlier in Equation 2. Intuitively, the edge weights measure the overhead due to conflicts in the LC when the nodes (traces) that share the edge are made to share LC chunks. The dynamic copies represent the minimum set of copies for a sample input. But in reality, the compiler may not be able to achieve this as it has to conservatively insert copies to ensure the legality constraint (see Section 3.3).

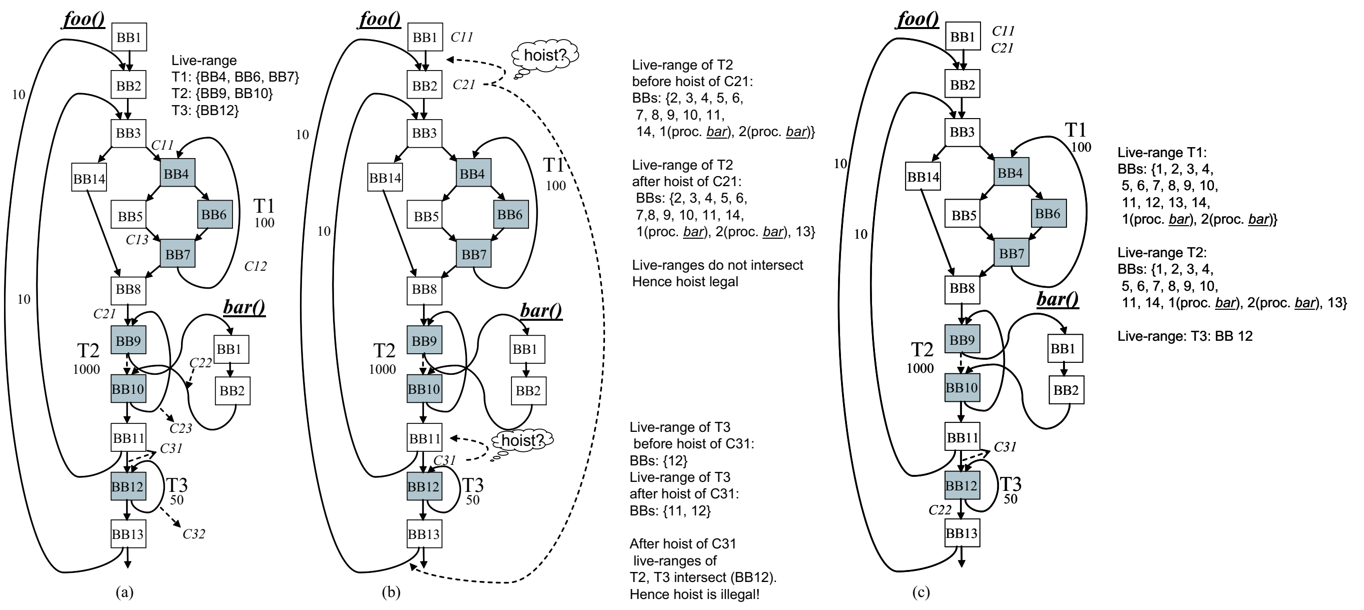
```
Sorted_Trace_List =
 Traces sorted in decreasing order of NetGain;
for each trace T in (Sorted_Trace_List) {
 benefit_found = false;
 for (c = 0; c < NUM_LC_CHUNKS; c++) {
  if (c+num_of_chunks(T) > NUM_LC_CHUNKS)
   break;
  Intersect_Traces =
    set of intersecting traces
    at LC chunks c to c + num_of_chunks(T);
  Dynamic_Copy_Cost =
    Compute_Copy_Cost(T, Intersect_Traces)
  net_benefit = Benefit(T) - Dynamic_Copy_Cost;
  if (net_benefit > curr_max_benefit) {
   curr_max_benefit = net_benefit;
   best_start_chunk = c;
   benefit_found = true;
  }
 }
 if (benefit_found)
  Place_Trace(T, best_start_chunk);
}
Compute_Copy_Cost(T, Intersecting_Traces)
{
 edge_wt = 0;
 for every Ti in Intersecting_Traces
  edge_wt += Edge_Weight(T, Ti, TRG);
 return edge_wt;
}
```

**Figure 5:** **Pseudo code to select and place traces in the LC.**

The placement algorithm uses the dynamic copy costs (edge weights in the TRG) to place each trace in the LC. The pseudo code for the trace selection/placement is shown in Figure 5. Traces are considered for placement in the decreasing order of gain ($NetGain > 0$). Each trace is considered at a particular chunk using *Compute_Copy_Cost* and is greedily placed at the LC index with the maximum *net_benefit*.

Figure 4(d) shows how T1, T2, and T3 are placed in the LC. Initially, since the LC is empty, T2, the highest benefit trace with *net_benefit* = *Benefit*, is placed at chunk 0. T1 is placed at chunk 2 where there is maximum *net_benefit* and zero interference. Since the LC is now full, T3 has to overlap with either T1 or T2. The dynamic copy costs when T3 overlaps with T2 and T1 (edge weights between T3-T1 and T3-T2) are shown on the right in Figure 4(d). Since both the choices have positive *net_benefit*s, there is an advantage in placing T3 in the LC. The *net_benefit* when T3 overlaps with T2 is higher than T1, hence T3 is placed at chunk 0 overlapping with T2.

It should be noted that traces are obtained for the whole pro-



Live-range
T1: {BB4, BB6, BB7}
T2: {BB9, BB10}
T3: {BB12}

Live-range of T2
before hoist of C21:
BBs: {2, 3, 4, 5, 6,
7, 8, 9, 10, 11,
14, 1(proc. bar), 2(proc. bar)}

Live-range of T2
after hoist of C21:
BBs: {2, 3, 4, 5, 6,
7,8, 9, 10, 11, 14,
1(proc. bar), 2(proc. bar), 13}

Live-ranges do not intersect
Hence hoist legal

Live-range of T3
before hoist of C31:
BBs: {12}
Live-range of T3
after hoist of C31:
BBs: {11, 12}

After hoist of C31
live-ranges of
T2, T3 intersect (BB12).
Hence hoist is illegal!

Live-range T1:
BBs: {1, 2, 3, 4,
5, 6, 7, 8, 9, 10,
11, 12, 13, 14,
1(proc. bar), 2(proc. bar)}

Live-range T2:
BBs: {1, 2, 3, 4,
5, 6, 7, 8, 9, 10,
11, 14, 1(proc. bar), 2(proc. bar), 13}

Live-range: T3: BB 12

**Figure 6: Copy placement example. (a) Initial copies inserted (b) Hoisting of copies and live-range computation (c) Final copy placement.**

gram. This allows selection and placement of traces across all procedures such that the conflicts are minimized. The placement heuristic is a greedy heuristic, giving preference to traces of highest benefit first. The greedy heuristic, though not an optimal solution, works well in practice. The selection and placement algorithm is similar to code placement techniques for cache miss rate reduction where the code is reorganized to minimize conflict misses and improve locality [26, 14].

## 3.3 Copy Placement

The goal of the copy placement phase is to insert *LC_COPY* instructions subject to the following issues:

- A selected trace should always be present in the LC when control enters the trace. If two traces T1 and T2 overlap in the LC, a copy of T2 could invalidate T1. Thus, after control leaves T2 and before T1 gets executed, T1 needs to be recopied.
- A copy of a trace should ideally occur only when it is needed. If a trace is already in the LC and has not yet been displaced, then it is pointless to recopy the trace. Thus, a copy should be inserted only when required. Since copies are stalling, redundant copies not only consume power, but also affect performance.

The copy placement algorithm handles the two issues using a phased approach. Initially, copies are inserted on all edges of the CFG that enter a trace. This naïvely guarantees that the trace is copied into the LC before execution regardless of which other traces displace it. Thus, this ensures correct but inefficient execution. Following this, a phase of iterative copy hoisting and redundant copy elimination is performed. Iterative copy hoisting attempts to hoist copies from their initial locations up the CFG, across procedure boundaries, to infrequently executed blocks subject to legality constraints. The legality constraint is that there should be sufficient copies to ensure that the trace is copied prior to execution if displaced by another overlapping trace. Figure 6(a) shows the initial location of copies in the CFG. We use the convention *Cij* to denote the $j^{th}$ copy for trace $i$. In the previous section, the trace selection/placement algorithm overlapped traces T2 and T3. Naïvely, if all copies for traces T2 and T3 are moved to BB1, then copies would overwrite each other. Thus without recopies, this would cause illegal execution. Also, since no other traces overlap T1, copies for T1, C12 and C13, are redundant. Hence, two of the three copies are removed. T1 requires just a single copy in BB1.

Copy insertion and hoisting are performed on a global CFG of the entire application, including all procedures, represented within the underlying IR. The global CFG connects all procedures with their call sites. For indirect calls, edges are drawn conservatively from the call site to all possible targets. Initial copies are placed on the edges of the CFG by creating extra pseudo BBs at the edges. In Figure 6(a), for sake of clarity, we show the initial copies on the edges.

Iterative copy hoisting and redundant copy elimination are done in the following steps.

**Live-Range Construction:** Intuitively, a trace needs to reside in the LC from the copy point until the point when control leaves the trace and never gets back to the trace. This is akin to live-ranges used in register allocation [8]. The live-range of a trace is defined as the set of blocks in the CFG starting from the point where the copy of the trace is defined until the last use of the trace.

To compute the live-range of a trace, we need to compute the blocks that are live-in to each trace and the blocks the copy reaches. Thus, traditional liveness and reaching-defs analysis [1] can be carried out for each trace to compute its live-range. Each trace identifier $T_i$ is modeled as a variable. The copy for a trace "defines" the trace and is assumed to be the first instruction in the BB where the copy is placed. The "use" of a trace includes all BBs that comprise the trace. For a given trace, there can only be one copy of that trace in a block. While for a given block, there can be multiple copies corresponding to different traces. The copy for a trace can "kill" another copy for the same trace. The initial live-ranges consists of just the blocks in the traces and are shown in Figure 6(a) for traces T1, T2, and T3. Although not shown in figure, the live-ranges also include the pseudo blocks on edges where the copies are present.

```
Eliminate_Redundant_Copies();
CopyQ = List of copies for all traces sorted in
        decreasing frequency order;
while (!CopyQ.is_empty()) {
 C = CopyQ.pop();
 bb = BB containing the C;
 PredSet = Predecessor BBs of bb
 while (!PredSet.is_empty()) {
  Remove C from bb
  Hoist copies to BBs in Pred_Set
  Compute_Live_Ranges()
  if (LiveRangesIntersect()) {
   undo hoist;
   CopyQ.remove(C) and finalize copy in bb;
   break;
  } else {
   Eliminate_Redundant_Copies();
   Old_Freq = freq of C;
   New_Freq = sum of frequencies of copies
              after C is hoisted;
   benefit = New_Freq - Old_Freq;
   if (benefit >= 0) {
    hoist is successful;
    insert the new copies in BBs contained in
      PredSet into the CopyQ;
    break;
   } else {
    PredSet = new set of predecessors of PredSet;
    }
   }
  }
 }
 CopQ.remove(C) and finalize copy in bb;
}
```

**Figure 7: Pseudo code to hoist and eliminate redundant copies.**

Once the live-ranges are constructed, the legality constraint can be defined as follows. If the live-range of two overlapping traces $T_i$ and $T_j$ intersect, then there is some path in the program flow where a copy of $T_i$ would displace $T_j$ before $T_j$ is recopied.

**Copy Hoisting:** The iterative hoisting algorithm tries to move copies that are in BBs that are frequently executed, up the CFG to BBs of lower frequencies while maintaining the legality constraint. The algorithm has two goals while hoisting copies : (a) reduce the overhead of executing the copies, and (b) ensure that the copies are present at the appropriate points in the program such that the traces are copied prior to their execution. The initial copy placement guarantees (b), but at the expense of executing copies even if the trace is not displaced by another overlapping trace.

These two goals conflict with each other. On the one hand, hoisting copies up the CFG to blocks of lower frequency is beneficial as it reduces the dynamic copy cost. But, on the other hand, the live-range of the trace corresponding to the copies grow longer as the copies are hoisted higher. This can interfere with the live-ranges of other traces that overlap with this trace, thus violating the legality condition. Alternately, it could prevent other copies from getting hoisted to ensure the legality condition. The copy hoisting algorithm addresses this problem by hoisting the most frequently executed copy only while it is the highest execution frequency. When its frequency decreases, hoisting is iteratively performed on the new highest frequency copy and so on.

The pseudo code for iterative copy hoisting is shown in Figure 7. The *Eliminate_Redundant_Copies* function (using dominator analysis), eliminates unnecessary copies of a given trace. A copy is redundant if the BB in which the copy is placed is dominated by another block which contains another copy for the same trace. The elimination includes a check for intersecting live-ranges for legality. In Figure 6(a), copy *C11* dominates copies *C12* and *C13*, hence *C12* and *C13* are eliminated. The rest of the copies for all traces are sorted in decreasing order of frequency. The hoisting algorithm picks the copy with the highest frequency and iteratively tries to hoist it to its predecessor blocks. If the live-range intersects with another overlapping trace, the copy is required at the current block and is not hoisted further. If the hoist is legal and the sum total of frequencies for the set of copies after the hoist is lower than the sum of the frequencies for the set of copies before the hoist, the algorithm can claim benefit and confirm the hoist.

Figure 6(b) illustrates the hoisting algorithm. Assume that the copies have been hoisted to the currently shown positions from those shown in Figure 6(a). C21 was moved from its home location on the edge from BB 8 to BB 9 (Figure 6(a)) to its current position in BB 2 (Figure 6(b)) which is outside the inner loop and hence of lesser frequency. It should be noted that in the example, copies can be hoisted to edges. Subsequently, BBs are instantiated at the edges (not shown in example) to house these copies. Assume that C11 has moved all the way up to the entry block BB1. Since T1 does not overlap with any other trace, this move is legal. C21 is the next copy for which hoisting is attempted to all incoming edges of its home block, BB2. The predecessors are the edges from BB1 to BB2 and the backedge from BB13 to BB2 (dotted lines). The live-ranges of T2 before and after the hoist of C21 are shown in Figure 6(b). The live-ranges do not intersect with other traces, thus the hoist is legal. Moreover, since the sum of the frequencies of the incoming edges is the same as the frequency of BB2, the hoist is considered beneficial. Next, C31, which is of the next highest priority, is hoisted from its home (edge from BB11 to BB12) to its predecessor block BB11. This causes the live-ranges of T2 and T3 to intersect. Since T2 and T3 overlap in the LC, this hoist is illegal.

The final copy placement and live-ranges are shown in Figure 6(c). The initial copies of traces T1 and T2 are performed in BB1. Before control enters T3, copy C31 is performed. After control leaves T3, T2 is copied back via copy C22. For a copy on an edge, a new basic block is created and inserted into the CFG. If a copy materializes on the return edge of the CFG (BB2 of *bar* to BB10 of *foo*), then the copy is inserted after the procedure call within the caller. While if a copy materializes on the call edge (BB9 of *foo* to BB1 of *bar*), it gets inserted before the procedure call in the caller.

**Discussion:** It should be noted that by using a global CFG, we are able to hoist copies across procedure boundaries. Considering each procedure independently restricts copy hoisting to the entry block of the procedure resulting in substantial copy overhead. Our original design was not inter-procedural and suffered large energy and performance penalties due to this problem. The global CFG also allows a copy to cross procedure calls, thus reducing the copy overhead significantly.

There are two alternatives that could be considered to handle the hoisting problem. The problem could be formulated as a form of code motion and use techniques like lazy code motion [18]. However, lazy code motion is not ideal because each instruction is positioned in sequence at the point of highest profitability, thereby giving one copy complete priority over others. More importantly, if the live-range intersections are not taken into account while hoisting, the legality condition can be violated. Conversely, by hoisting the most profitable copy all the way up, its live-range is increased. This can prevent the hoisting of other copies as it would intersect with the live-range of the trace corresponding to the hoisted copy. Thus, interactions between multiple copies must be considered.

Alternately, one could place the copies in the prologue block of the 'main' procedure. This would cause the live-ranges of all traces to intersect. These live-ranges could then be 'split' by inserting copies at less costly points. But, live-range splitting heuristics employed in register allocation focus on reducing the register pressure so that a later coloring phase can allocate the live-ranges with reduced spill [7]. In our case, the traces have already been selected and placed if necessary in overlapping chunks in the LC. The copy placement and hoisting must

| size (bytes) | read (nJ) | write (nJ) |
|---|---|---|
| 32 | 0.0506 | 0.0388 |
| 64 | 0.0527 | 0.0413 |
| 128 | 0.0568 | 0.0463 |
| 256 | 0.0651 | 0.0563 |
| 512 | 0.0698 | 0.0701 |
| 1024 | 0.0990 | 0.1174 |
| 2048 | 0.1020 | 0.1228 |
| 4096 | 0.1197 | 0.1416 |

| size (bytes) | fetch (LC) (nJ) | fetch (Icache) (nJ) |
|---|---|---|
| 64 | 0.1803 | 0.2961 |
| 128 | 0.1888 | 0.3059 |
| 256 | 0.1980 | 0.4732 |
| 512 | 0.2188 | 0.4966 |
| 1024 | 0.2404 | 0.5233 |
| 2048 | 0.2748 | 0.5655 |
| 4096 | 0.3277 | 0.6351 |

**Table 1: Per access LC energy for the WIMS processor (top) and per access LC and icache energy using CACTI (bottom, .18$\mu$m) for different sizes.**

guarantee the legality of the placement by introducing appropriate 'spills' (recopies) at reduced copy overhead.

## 4. EXPERIMENTAL EVALUATION

### 4.1 Methodology

Our experimental framework consists of a port of the Trimaran compiler system [27] to the WIMS processor and a modified version of the WIMS processor simulator to model a parameterized LC. Note that since the WIMS processor is single-issue, many of the VLIW transformations, except function inlining, in Trimaran were disabled for these experiments. The simulator uses a simple energy model attributing a fixed energy to each LC or memory access for instruction fetch and totaling it up across the run of an application. On the WIMS processor, instruction fetch energy accounts for approximately 30% of the overall system energy including the processor and memory. For this study, a set of embedded benchmarks selected from the MediaBench and MiBench suites were chosen. For all experiments, compiler-directed dynamic placement (dynamic for short) is compared with compiler-directed static placement (static for short) that uses profile information to maximally pack the most frequently executed regions into the LC. In addition, a comparison against varying sizes of traditional instruction caches (icache for short) was also performed. For each of the experiments, two measures are presented. First, the instruction fetch energy consumption of each technique with respect to the baseline where all instructions are fetched from main memory. Second, the hit rate is the ratio of accesses (LC or icache) to total instruction references.

Two memory configurations were used - WIMS and CACTI. For the WIMS processor, LC size was varied from 32 to 4k bytes for the study. With each LC size, the energy presented in Table 1 (top) was assumed for each access. The energy consumption estimates were obtained from configuration specific data sheets from a popular memory compiler for a $0.18\mu$m process. These values compare to 0.1384 nJ per read of a 16Kb bank from the regular on-chip memory. The access times for the on-chip main memory and LC are both one cycle.

To compare against an icache of equal size, CACTI [32] was used to obtain the energy numbers for different sizes of icache and LCs. Here, the icache and LC are assumed to be on-chip, while the main memory is off-chip. For LC, the energy for the tag and comparator circuits were subtracted from a correspondingly sized direct-mapped icache [4]. The LC and icache sizes were varied from 64 to 4k [1]. For the icache, we assumed 16-byte line size with 2-way associativity to get the power numbers as shown in Table 1 (bottom). The Am41PDS3228D SRAM [2] was assumed to be the off-chip memory with 3.024nJ per access (16-bits). The icache hit rates were obtained using the Dinero-IV cache simulator [12]. While comparing static and dynamic with icache, all measurements were obtained using the CACTI power numbers. For comparing static versus dynamic for the WIMS processor, all measurements were obtained relative to the WIMS power model. For all studies, a single read/write from the cache to the main memory is assumed to be 2-bytes. In Table 1, although there is a difference in absolute energy values between the WIMS and the CACTI models, this is less important as we do relative comparisons within each class. The difference in energy numbers between WIMS and CACTI is due to an abstract energy model used by CACTI as opposed to real datasheet numbers used for WIMS.

### 4.2 Results

**Comparison with static:** The energy savings and LC hit rates for static and dynamic placement are compared across all the benchmarks for LC sizes of 64 and 256 bytes in Figure 8 for the WIMS processor. Considering first the 64 byte LC, dynamic is generally more effective at utilizing the LC. The largest energy benefits occur for cjpeg, unepic, and sha where the static placement savings are more than doubled with dynamic placement. These benchmarks achieve such large gains by increasing the hit rates in the LC by similar amounts due to more effective utilization of the LC. The epic application achieves the largest total energy savings of 58% with dynamic placement. However, static placement is also very successful with this benchmark, achieving 42%. Epic has a relatively small innermost loop where a large fraction of the execution time is spent, and the entire loop body can be placed in the 64-byte LC. Dynamic achieves a modest gain above that by relocating another loop body into the LC. Overall, dynamic placement achieves an average energy savings of 28% across the benchmarks compared with 17% for static placement.

Examining the 256-byte LC graphs, the energy savings and hit rates achieved with static and dynamic placement are much closer. Clearly, as the LC size is increased, the importance of dynamic placement goes down as a larger fraction of the hot regions statically fit into the LC. Cjpeg, djpeg, unepic, gsmencode, mpeg2enc, and pgpencode are examples where dynamic is still very effective as these benchmarks have a large memory footprint. A small fraction of benchmarks, where the energy savings with dynamic placement exceeded static for the 64-byte LC, now achieve worse results with the 256-byte LC. Examples of this behavior are g721encode, g721decode, rawcaudio, and rawdaudio. These benchmarks are characterized by a modest number of conflicts between LC entries. The copies could not be hoisted out of frequently executed code regions due to interference, thus a large number of run-time copies must be performed. Rawcaudio and rawdaudio have small code size; thus, static is able to pack all the hot regions in the application into the LC without any run-time penalty. Dynamic, on the other hand, achieves the same hit-rate but at the expense of the one-time copy overhead. Overall, for both 64 and 256 byte LC configurations, by packing multiple hot regions, dynamic is effective at increasing LC hit rates.

The effect of varying LC size on four representative benchmarks on the WIMS processor is shown in Figure 9. Each graph contains 4 lines: LC hit rate for dynamic, LC hit rate for static, energy savings for dynamic, and energy savings for static. Note that hit rates (shaded light) use the left hand y-axis and energy (shaded dark) use the right hand y-axis. A number of interesting trends can be observed from these graphs. First, at smaller

[1]CACTI did not support a 32-byte cache

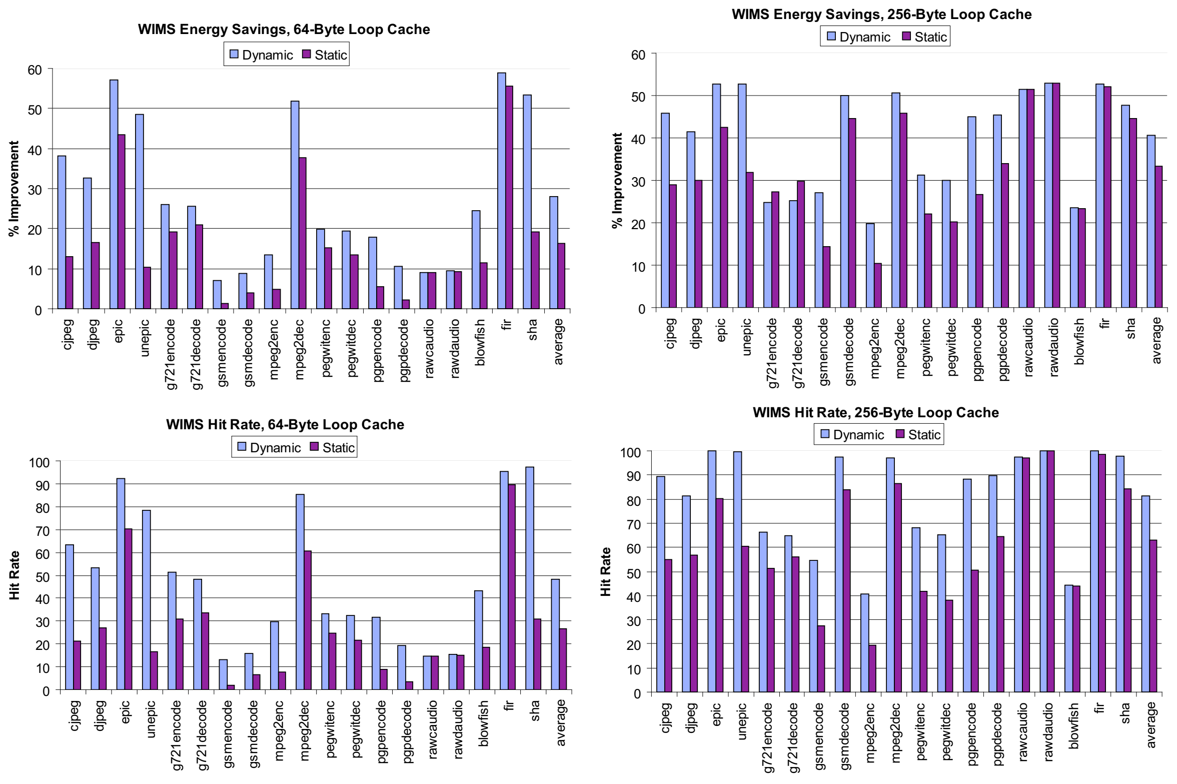


**Figure 8:** **Comparing energy savings and hit rate of static and dynamic over on-chip main memory for the WIMS processor.**

LC sizes, dynamic placement outperforms static placement by a large margin. For mpeg2dec, dynamic placement increases energy savings from 35% to 75% for a 32-byte LC and from 60% to 80% for a 64-byte LC. Similarly for pgpdecode, energy savings increases from 38% to 92% for a 128-byte LC. For the other benchmarks, the differences are not as large, but the same trend occurs. The reason for the increased energy savings is the ability of dynamic placement to increase LC utilization. Most of these applications contain a number of hot code regions that collectively cannot fit in the LC using static placement. It is thus critical to relocate different regions of code into the LC at different points during program execution to take full advantage of the LC. The increased utilization is evident by the large increase in hit rate of dynamic over static for the smaller LC sizes.

A second trend seen in all the graphs is that energy savings goes down for larger LC sizes, particularly the 1k, 2k, and 4k configurations. The peak energy savings comes at around 128-512 byte LCs. The reason for this behavior is two fold. First, it becomes less beneficial to relocate instructions with larger LCs. For larger LCs, the energy characteristics are close to that of the on-chip memory, thus the potential savings becomes less. Second, the overhead of dynamic copying becomes larger, thereby taking away from the percentage savings. For the larger LC sizes, static performs a good job of packing a significant fraction of the hot code without any overhead. For dynamic, copy overhead causes the energy savings to depreciate.

**Comparison with icache:** Figure 10 shows the energy and hit rates for static, dynamic, and icache for 64 and 256-byte on-chip cache configurations using CACTI energy models. The average energy savings for both static and dynamic are much higher than the WIMS processor. For the 256 byte LC, static achieves 59%, while dynamic achieves 79% average energy savings. This is largely due to the costlier off-chip memory access. The off-chip main memory is over 20x more power hungry than the on-chip memory.

For all the benchmarks, icache is able to get higher hit-rates compared to static and dynamic schemes. On average, we observe 98%, 82%, and 62% hit-rate for icache, dynamic, and static respectively in the 256-byte cache configuration. In the 64-byte case, icache records a 70% improvement in hit-rate over dynamic. But this improvement comes at the expense of energy. Each access to the icache requires tag checks and hence is costlier than the tag-less LC. In addition, a miss for icache is much more expensive, as it has to fetch a cache-line of instructions (16-bytes) into the icache every time, which involves multiple accesses to both the main memory and the icache.

The dynamic scheme is geared towards reducing the overall energy as opposed to raw hit-rate. The dynamic scheme copies a flexible number of chunks using a LC-COPY only when deemed beneficial by the compiler, thus leading to a more power efficient LC utilization. A miss requires only a single access from the main memory. Dynamic is able to achieve 66% and 33% improvement in energy savings over icache for the 64-byte and 256-byte LC configurations respectively. In the 64-byte case, for rawcaudio, rawdaudio, pgpdecode, and gsmdecode, although icache registers over 70% hit rate, there is a decrease in energy savings over main memory. Overall, icache performs better than static, while dynamic performs better than the two.

**Detailed comparisons:** Table 2 (left) compares the code size and the size of the cache required for static, dynamic, and icache to cover 95% of the dynamically executed code for each benchmark. On average, all cache configurations require less than 10x the code size, which verifies the 90-10 rule. More interestingly, dynamic requires 2.5x less cache size than static, while icache, with a higher hit rate, requires 7.6x less cache than static. Table 2 (middle) shows the LC size for maximum energy savings on the WIMS processor for each benchmark. On average, the LC size of dynamic is 1.68x less than the static scheme. Dy-

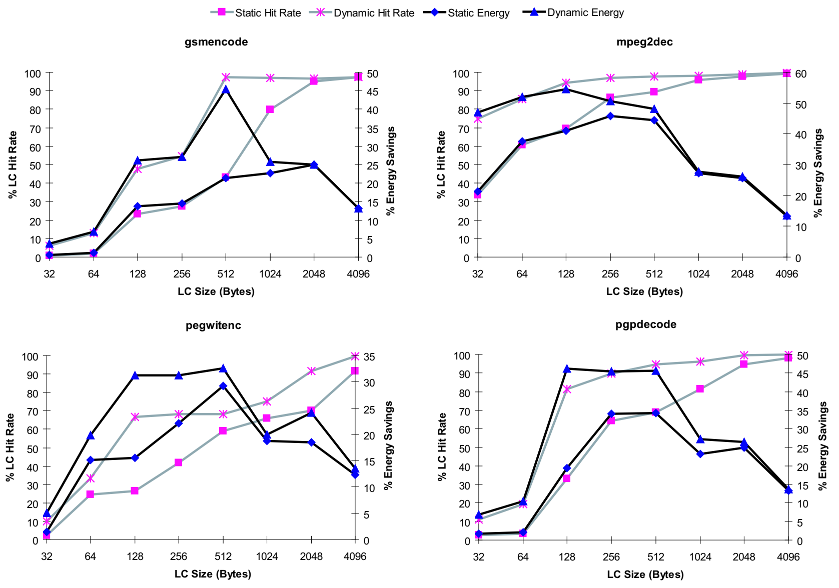


**Figure 9: Effect of varying LC size on energy savings and hit rate over on-chip main memory for the WIMS processor.**

| benchmark | size(Kb) | static (b) | dynamic (b) | icache (b) |
|---|---|---|---|---|
| cjpeg | 63 | 4096 | 1024 | 256 |
| djpeg | 68 | 2048 | 512 | 256 |
| epic | 11 | 1024 | 128 | 64 |
| unepic | 13 | 1024 | 128 | 128 |
| g721encode | 4 | 2048 | 2048 | 256 |
| g721decode | 4 | 2048 | 2048 | 512 |
| gsmencode | 21 | 2048 | 512 | 512 |
| gsmdecode | 19 | 1024 | 256 | 128 |
| mpeg2enc | 35 | 4096 | 1024 | 512 |
| mpeg2dec | 24 | 1024 | 256 | 64 |
| pegwitenc | 21 | 8192 | 4096 | 256 |
| pegwitdec | 21 | 4096 | 2048 | 512 |
| pgpencode | 102 | 4096 | 512 | 256 |
| pgpdecode | 102 | 4096 | 512 | 256 |
| rawcaudio | .8 | 256 | 256 | 256 |
| rawdaudio | .8 | 256 | 256 | 256 |
| blowfish | 5 | 1024 | 1024 | 1024 |
| fir | .5 | 256 | 64 | 64 |
| sha | 1 | 512 | 64 | 64 |
| **average** | 27.16 | 2277.05 | 882.53 | 296.42 |

| benchmark | static | dynamic |
|---|---|---|
| cjpeg | 512 | 256 |
| djpeg | 512 | 512 |
| epic | 512 | 128 |
| unepic | 512 | 128 |
| g721encode | 512 | 512 |
| g721decode | 512 | 512 |
| gsmencode | 2048 | 512 |
| gsmdecode | 512 | 128 |
| mpeg2enc | 512 | 512 |
| mpeg2dec | 256 | 128 |
| pegwitenc | 512 | 512 |
| pegwitdec | 512 | 512 |
| pgpencode | 512 | 128 |
| pgpdecode | 512 | 128 |
| rawcaudio | 256 | 256 |
| rawdaudio | 256 | 256 |
| blowfish | 1024 | 1024 |
| fir | 64 | 64 |
| sha | 512 | 64 |
| **average** | 502.86 | 298.87 |

| benchmark | size | % gain |
|---|---|---|
| cjpeg | 64 | 25.13 |
| djpeg | 64 | 16.02 |
| epic | 128 | 14.26 |
| unepic | 64 | 38.17 |
| g721encode | 32 | 12.46 |
| g721decode | 32 | 10.83 |
| gsmencode | 512 | 24.06 |
| gsmdecode | 128 | 9.35 |
| mpeg2enc | 128 | 12.19 |
| mpeg2dec | 32 | 25.79 |
| pegwitenc | 128 | 15.68 |
| pegwitdec | 128 | 15.35 |
| pgpencode | 128 | 30.89 |
| pgpdecode | 128 | 26.79 |
| rawcaudio | 128 | 0.10 |
| rawdaudio | 64 | 0.06 |
| blowfish | 64 | 13.03 |
| fir | 32 | 19.18 |
| sha | 64 | 34.26 |
| **average** | 110.22 | 17.69 |

**Table 2: The left table shows the code cache size (in bytes) for at least 95% hit rate for static, dynamic, and icache schemes for the CACTI energy models. The middle table shows the LC size required for highest energy gains in the WIMS processor. The right table shows the LC size at maximum energy gain of dynamic over static.**

namic consumes more cache space than icache as it tries to reduce the overlaps of traces in the LC so as to decrease the recopy overhead. Table 2 (right) give the size of the LC for maximum energy gains of dynamic over static. On average, at 110 bytes, dynamic shows over 17% improvement over static.

Finally, Table 3 quantifies the LC access behavior of the dynamic allocation scheme for a 256-byte LC. For each benchmark, column 2 gives the total number of traces in the code that were allocated to the LC. Column 3 shows the same metric expressed as a percentage of all the traces in the code. For example, for pgpencode, 155 traces were packed into the LC, which accounted for only 2.2% of all the traces in the code. Although not shown in the table, the 155 traces had a combined size of 5186 bytes, which is 20x the size of the 256-byte LC, but 5% of the total code size. Column 4 gives the dynamic execution frequency for these selected static traces. For pgpencode, 2.2% of the hot traces accounted for 87% of the dynamic execution frequency. Column 5 gives the mean number of trace overlaps per LC chunk. Note that a trace can overlap multiple chunks. For a 256-byte LC, there are 16 chunks assuming 16-bytes per chunk. Again for pgpencode, an average of 24.5 traces were overlapped per chunk. The dynamic scheme was able to successfully pack the most frequent hot traces and overlap them with the least number of conflicts. Finally, column 6 gives the percent overhead in cycles due to dynamic copying. Although the dynamic placement algorithm was driven by energy constraints, the performance degradation is only marginal, with an average of 2.57% loss due to copy stalls.

The dynamic placement algorithm was based on profiling the applications on a sample input. Although our final statistics were compiled using the same input, we did validate, by running on different input sets, that the control flow behavior of the benchmarks did not change significantly. The resulting performance/energy changed by less than 1%.

## 4.3 Comparison to ILP-based solutions

While [25, 31] address compiler-directed dynamic placement, we believe the proposed ILP-formulation for optimal placement is impractical for moderate to large sized applications. The ILP-solution attempts to model the problem by having variables for all possible traces at every edge on the CFG. Each such variable denotes whether the trace is placed in the LC or memory, and whether it needs to be recopied or not. The problem is formulated based on an earlier work on ILP-based optimal register

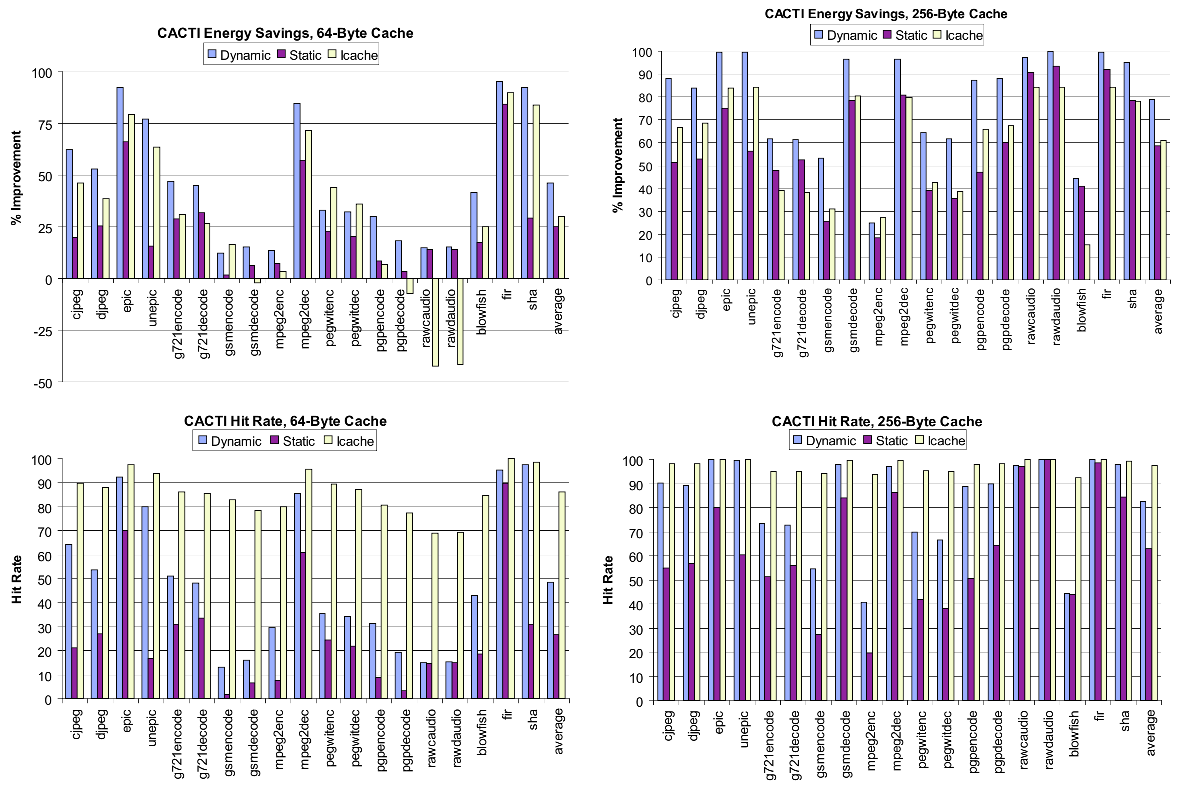


**Figure 10: Comparing energy savings and hit rate of static, dynamic, and icache over off-chip main memory using CACTI.**

allocation [3]. This works well as long as the number of constraints and variables are limited, but explodes when attempted for full inter-procedural analysis of large programs. Table 4 reflects our implementation of [31] and shows the number of variables and constraints, the time taken to solve for inter-procedural placement using a commercial ILP-solver, CPLEX, and the percentage energy improvement over the dynamic scheme. Such long run-times for a large number of variables have been acknowledged by the authors in their paper [31]. [25] and [31] try to reduce this complexity by limiting their analysis to within a procedure or loop boundaries. But this causes the LC contents to be flushed after procedure calls/returns. We observed excessive redundant copies to restore the LC contents without inter-procedural analysis.

Our heuristic based solution offers an alternative approach to ILP that does not achieve optimal results, but is practical and can handle full inter-procedural analysis of large programs with arbitrary control flow. For smaller loop-dominated benchmarks with multiple loop-nests and simple control-flow, the dynamic scheme was able to select and overlap an optimal set of traces. The iterative copy hoisting algorithm was successful in positioning the copies at optimal points in the code. As seen in Table 4, for all benchmarks the dynamic scheme performed close to optimal. The degradation for dynamic was observed due to the greedy nature of the trace placement and copy hoisting heuristics.

## 5. CONCLUSION

In this paper, we have proposed an approach for compiler-directed dynamic placement of instructions into a low-power code cache. Dynamic placement enables the compiler to use entries in the code cache to hold multiple hot code regions (traces) over the execution of an application, thereby increasing the code cache utilization by a substantial amount. These traces can have any complex control flow or embedded procedure calls. Dynamic placement is accomplished in two major steps. First, the code cache entries are allocated among all the candidate traces. A heuristic cost/benefit analysis compares the expected copy cost with the anticipated energy benefit. Second, copies are inserted followed by an iterative process of hoisting and redundancy elimination using liveness analysis on a inter-procedural control flow graph to derive a cost-effective placement. Our investigation was carried out in the context of the WIMS microcontroller, a processor designed for embedded sensor systems where power consumption is the dominant design concern. Results show an average energy savings of 28% with dynamic placement compared with 17% with static for a 64-byte code cache. This is accomplished by increasing the code cache hit

| benchmark | # variables | # equations | time | % gain |
|---|---|---|---|---|
| fir | 2034 | 2446 | 1 min | 0 |
| rawcaudio | 1980 | 2516 | 1 min | 2 |
| rawdaudio | 1404 | 1778 | 1 min | 0 |
| g721encode | 2927 | 2584 | 1 min | 11 |
| g721decode | 3238 | 2950 | 1 min | 4 |
| blowfish | 8528 | 10169 | 1 min | 4 |
| sha | 13598 | 14773 | 3 min | 2 |
| gsmencode | 148538 | 142259 | 20 min | 6 |
| gsmdecode | 150772 | 143394 | 20 min | 2 |
| epic | 420170 | 515835 | 1 hr | 1 |
| unepic | 227852 | 298993 | 1 hr | 1 |
| cjpeg | 1210512 | 1496334 | 40 hr | 3 |
| djpeg | 933002 | 1149342 | 40 hr | 3 |
| pegwitenc | 1732012 | 2093478 | 60 hr | 4 |
| pegwitdec | 1565644 | 1904755 | 60 hr | 4 |
| mpeg2enc | 7666254 | 9850505 | * | * |
| mpeg2dec | 3534664 | 4382525 | * | * |
| pgpencode | 12673300 | 16033319 | * | * |
| pgpdecode | 10404038 | 13488579 | * | * |

**Table 4: Size, time taken, and percent energy gain over the dynamic scheme for a full program ILP-formulation using CPLEX on a 1-GHz UltraSPARC-IIIi processor. '*' denotes failure to complete within 72 hours of run-time.**

| benchmark | # traces | % traces | % dynamic. freq. | overlaps | % perf loss |
|---|---|---|---|---|---|
| cjpeg | 79 | 1.65 | 89.72 | 13 | -2.11 |
| djpeg | 58 | 1.11 | 80.36 | 8.81 | -2.17 |
| epic | 60 | 7.25 | 99.74 | 8.12 | -1.11 |
| unepic | 68 | 6.58 | 99.89 | 11.31 | -1.06 |
| g721encode | 9 | 3.11 | 63.11 | 1.69 | -8.55 |
| g721decode | 7 | 2.35 | 61.85 | 1.44 | -7.56 |
| gsmencode | 45 | 3.66 | 58.88 | 9.38 | -2.31 |
| gsmdecode | 14 | 1.04 | 97.58 | 3.31 | -2.19 |
| mpeg2enc | 244 | 13.62 | 41.81 | 31.25 | -2.39 |
| mpeg2dec | 51 | 3.16 | 96.32 | 7.19 | -1.56 |
| pegwitenc | 42 | 4.65 | 72.15 | 7.31 | -4.68 |
| pegwitdec | 45 | 4.89 | 69.42 | 8.31 | -4.37 |
| pgpencode | 155 | 2.20 | 87.13 | 24.5 | -2.21 |
| pgpdecode | 122 | 1.75 | 89.02 | 20.19 | -2.58 |
| rawcaudio | 8 | 10.13 | 97.77 | 1.06 | -0.01 |
| rawdaudio | 8 | 10.39 | 99.96 | 1.00 | -0.01 |
| blowfish | 10 | 10.31 | 50.14 | 1.62 | -1.05 |
| fir | 11 | 33.33 | 99.84 | 1.31 | -1.17 |
| sha | 9 | 15.25 | 98.24 | 1.44 | -4.21 |
| **average** | 55.00 | 7.18 | 81.73 | 8.54 | -2.57 |

**Table 3: Benchmark characteristics on a 256-byte LC for dynamic allocation showing number of hot traces selected, fraction of the total number of traces, dynamic execution frequency, average number of trace overlaps per chunk in LC, and percent performance degradation due to copy overhead.**

rate from an average of 26% to 49%. For a 256-byte code cache, a more modest increase in energy savings occurs; 41% with dynamic verses 32% with static. In comparison to a traditional instruction cache, dynamic placement achieves an average of 25% energy savings.

## 6. ACKNOWLEDGMENTS

We thank Manish Verma, University of Dortmund for his help with the ILP formulation. Fabrication of this work at TSMC was supported by the MOSIS Educational Program. Digital cell libraries and SRAMs were supplied by Artisan Components, Inc. This work was supported by the Engineering Research Centers Program of the National Science Foundation under award number EEC-9986866, NSF grant CCF-0347411, and equipment donated by Hewlett-Packard and Intel Corp.

## 7. REFERENCES

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison Wesley, 1985.

[2] AMD. *Am41PDS3228D SRAM*, 2004. http://www.amd.com/us-en/FlashMemory/ProductInformation/.

[3] A. Appel and L. George. Optimal spilling for cisc machines with few registers. In *Proc. of Programming Language Design and Implementation*, Jun. 2001.

[4] R. Banakar et al. Scratchpad memory: A design alternative for cache on-chip memory in embedded systems. In *Proc. of 10th Intl. Symposium on Hardware/Software Codesign*, May 2002.

[5] N. Bellas et al. Energy and Performance Improvements in Microprocessor Design Using a Loop Cache. In *Proc. of International Conference on Computer Design*, Oct. 1999.

[6] L. Benini et al. Asymptotic Zero-Transition Activing Encoding for Address Busses in Low-Power Microprocessor-Based Systems. In *IEEE Great Lakes Symposium on VLSI*, Mar. 1997.

[7] P. Briggs. Register allocation via graph coloring. Technical Report TR92-183, 24, 1992.

[8] G. Chaitin. Register allocation and spilling via graph coloring. In *Proc. of SIGPLAN Symp. on Compiler Construction*, Jun. 1982.

[9] P. Chang and W. Hwu. Trace selection for compiling large C application programs to microcode. In *Proc. of 21st Intl. Symposium on Microarchitecture*, Nov. 1988.

[10] S. Cotterell and F. Vahid. Tuning Loop Cache Architectures to Programs in Embedded System Design. In *Proc. of Internationl Symposium on System Synthesis*, Oct. 2002.

[11] D. Dobberpuhl. The design of a high-performance low-power microprocessor. In *Proc. of Intl. Symposium on Low Power Electronics and Design*, Aug. 1996.

[12] J. Elder and M. D. Hill. Dinero IV Trace-Driven Uniprocessor Cache Simulator.

[13] J. A. Fisher. Trace scheduling: A technique for global microcode compaction. *IEEE Transactions on Computers*, July 1981.

[14] N. Gloy et al. Procedure Placement Using Temporal Ordering Information. In *Proc. of 30th Intl. Symposium on Microarchitecture*, Dec. 1997.

[15] A. Gordon-Ross et al. Exploiting fixed programs in embedded systems : A loop cache example. *Comp. Arch. Letters*, 1(1), 2002.

[16] N. Kim et al. Circuit and Microarchitectural Techniques for Reducing Cache Leakage Power. *IEEE Trans. on VLSI*, 12(2), Feb. 2004.

[17] J. Kin et al. The Filter Cache: An Energy Efficient Memory Structure. In *Proc. of 30th Intl. Symposium on Microarchitecture*, Dec. 1997.

[18] J. Knoop et al. Lazy code motion. In *Proc. of Programming Language Design and Implementation*, June 1992.

[19] L. Lee et al. Low-Cost Embedded Program Loop Caching - Revisited. Technical Report CSE-TR-411-99, Univ. of Michigan.

[20] L. Lee et al. Instruction Fetch Energy Reduction Using Loop Caches for embedded applications with Small Tight Loops. In *Intl. Symp. on Low Power Electronics and Design*, Aug. 1999.

[21] H. Lekatsas et al. Code Compression for Embedded Systems. In *Proc. 35th Design Automation Conference*, Jun. 1998.

[22] P. R. Panda et al. *Memory Issues in Embedded Systems-On-Chip*. Kluwer Academic Publishers, Norwell, MA, 1999.

[23] R. M. Senger et al. A 16-Bit Mixed-Signal Microsystem with Integrated CMOS-MEMS Clock Reference. In *Procedings of the Design Automation Conference*, Jun. 2003.

[24] S. Steinke et al. Assigning Program and Data Objects to Scratchpad for Energy Reduction. In *Proc. of Intl. Conference on Design, Automation and Test in Europe*, Mar. 2002.

[25] S. Steinke et al. Reducing energy consumption by dynamic copying of instructions onto onchip memory. In *Proc. of Intl. Symposium on System Synthesis*, Oct. 2002.

[26] H. Tomiyama and H. Yasuura. Code Placement Techniques for Cache Miss Rate Reduction. *ACM Transactions on Design Automation of Electronic Systems*, 2(4), Oct. 1997.

[27] Trimaran. An Infrastructure for Research in ILP. http://www.trimaran.org.

[28] S. Udayakumaran and R. Barua. Compiler-decided dynamic memory allocation for scratch-pad based embedded systems. In *Proc. of Intl. Conf. on Compilers Architectures and Synthesis of Embedded Systems*.

[29] Uh et al. Techniques for Effectively Exploiting a Zero Overhead Loop Buffer. In *Proc. of Compiler Construction*, 2000.

[30] T. VanderAa et al. Instruction buffering exploration for low energy VLIWs with instruction clusters. In *Proc. of Asia and South Pacific Design Automation Conference*, Jan. 2004.

[31] M. Verma et al. Dynamic overlay of scratchpad memory for energy minimization. In *Proc. of Intl. Symposium on System Synthesis*, Sept. 2004.

[32] S. Wilton et al. CACTI: An enhanced cache access and cycle time model. *IEE Journal of Soild State Circuits*, 31(5), May. 1996.