Energy-Efficient Program Layout for Multi-bank Memory Architectures

by

Nan Li

(Under the direction of Scott Watterson)

Abstract

Energy conservation is an important problem for battery-powered embedded or portable systems. New technology such as RDRAM enables memory to operate at different power levels. This allows memory to be partitioned such that only the necessary parts of memory are in active mode, while the others are in low power modes. The traditional program layout has code and heap at one end, and stack at the other end. However, it is possible to place the code in the high address range (next to the stack) without loss of functionality. This thesis explores the energy impact of those two layouts in a partitioned power aware memory system and presents a static program analysis technique to predict the more energy-efficient layout for a given program. We verify the effectiveness of our analysis by running MiBench programs on an enhanced Simplescalar-based power simulator. Our experimental results show that the new layout saves up to 43% of the memory subsystem energy when compared to the traditional layout, with an average improvement of 12%, on a cache-less CPU (such as the widely used ARM7TDMI). Our static analysis correctly predicts 13 out of 15 benchmarks from the MiBench suite. We also evaluate our scheme on a processor with several cache configurations. The cached configurations benefit much less (averaging less than 1%), though some programs see as much as a 25% energy savings from the non-traditional layout.

Index words: low power, compiler, static analysis, code layout, RDRAM
ENERGY-EFFICIENT PROGRAM LAYOUT FOR MULTI-BANK MEMORY
ARCHITECTURES

by

NAN LI

B.E., Shanghai Jiao Tong University, 2000

A Thesis Submitted to the Graduate Faculty
of The University of Georgia in Partial Fulfillment
of the
Requirements for the Degree

MASTER OF SCIENCE

ATHENS, GEORGIA

2003
Energy-Efficient Program Layout for Multi-bank Memory Architectures

by

Nan Li

Approved:

Major Professor: David Lowenthal
Committee: Maria Hybinette Kang Li

Electronic Version Approved:

Maureen Grasso
Dean of the Graduate School
The University of Georgia
December 2003
Acknowledgments

First and foremost, I want to thank Professor Scott Watterson for directing me through all the research phases including research topic selection, project implementation, experiments and paper writing. Without him, this work will not be done.

I also thank Professor David Lowenthal for teaching me operating systems and parallel programming. In his class, I learned how to critically think of research work. This is very valuable for me. I also miss the time while working as a teaching assistant for Professor Maria Hybinette. My little knowledge of parallel simulation is from her. I especially thank Professor Kang Li for giving me the opportunity to practice systems administration. I learned a great deal of Linux and networking skills through this. In addition, I must thank them for serving on my committee.

My thanks also go to Professor Suchendra M. Bhandarkar, my former research supervisor. He taught me a lot about computer vision and gave me research opportunities in my first year. Professor Surendar Chandra guided me into the computer networking field. I am grateful for his encouragement when I was unclear about the future.

Finally, I owe too much to my family and Ms. Jing Shen. I truly appreciate their enduring patience and support.
## Table of Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Acknowledgments</td>
<td>iv</td>
</tr>
<tr>
<td>List of Figures</td>
<td>vii</td>
</tr>
<tr>
<td>List of Tables</td>
<td>ix</td>
</tr>
<tr>
<td>Chapter</td>
<td></td>
</tr>
<tr>
<td>1 Introduction</td>
<td>1</td>
</tr>
<tr>
<td>2 Layout Prediction</td>
<td>3</td>
</tr>
<tr>
<td>2.1 Static Analysis</td>
<td>3</td>
</tr>
<tr>
<td>2.2 Memory Reference Estimation</td>
<td>9</td>
</tr>
<tr>
<td>2.3 Prediction</td>
<td>10</td>
</tr>
<tr>
<td>2.4 Example</td>
<td>10</td>
</tr>
<tr>
<td>3 Experimental Architectures</td>
<td>13</td>
</tr>
<tr>
<td>3.1 Processors</td>
<td>13</td>
</tr>
<tr>
<td>3.2 Memory Subsystem</td>
<td>14</td>
</tr>
<tr>
<td>3.3 Simulator</td>
<td>15</td>
</tr>
<tr>
<td>4 Experimental Results</td>
<td>17</td>
</tr>
<tr>
<td>4.1 Static Prediction Evaluation</td>
<td>17</td>
</tr>
<tr>
<td>4.2 Stack Register Analysis Evaluation</td>
<td>19</td>
</tr>
<tr>
<td>4.3 Energy Savings</td>
<td>20</td>
</tr>
<tr>
<td>4.4 Execution Time</td>
<td>25</td>
</tr>
</tbody>
</table>
5 Related work ........................................ 29

6 Conclusions ......................................... 33

Bibliography ........................................... 35
# List of Figures

2.1 Pseudo-code for block weight computation ........................................ 5
2.2 Pseudo-code for stack register analysis in a block ............................... 7
2.3 Pseudo-code for stack register analysis in a function ............................ 8
2.4 Pseudo-code for memory reference computation (Note loopWeight and recursiveWeight can be different between user function and library function) ................................................................. 9
2.5 Simple example C code ........................................................................ 10
2.6 Control flow graph for the code shown in Figure 2.5. Loop blocks are shown in gray, stack instructions are marked with an S, and heap instructions are marked with an H. Instructions marked with C accesses code memory, which contains a pointer to the static data in this case .............................................................................. 11

4.1 heap/stack reference ratios. We predict correctly if both ratios are above or below 1 at the same time ................................................................. 18
4.2 The percentage of predicted heap instructions that are indeed heap instructions in the simulation. ................................................................. 20
4.3 The percentage of predicted stack instructions that are indeed stack instructions in the simulation. ................................................................. 21
4.4 Energy savings due to CTP controlled RDRAM using the traditional layout, compared to a static system with conventional DRAM, which is always in high power mode. No cache is involved. ....................... 22
4.5 Energy savings of our predicted code layout over traditional layout with no cache ........................................ 23
4.6 Energy Increase of modified code layout over traditional layout with no cache ........................................ 24
4.7 Energy savings of our predicted code layout over traditional layout with 4K I-cache and 4K D-cache .................. 25
4.8 Energy savings of our predicted code layout over traditional layout with 8K unified cache .......................... 26
4.9 Execution slowdown due to CTP controlled RDRAM using our predicted code layout and traditional layout. Both are compared to a static system with conventional DRAM. No cache is used. ........ 27
4.10 Execution slowdown due to CTP controlled RDRAM using our predicted code layout and traditional layout. Both are compared to a static system with conventional DRAM. 4K I-cache and 4K D-cache are used. ................................................................. 27
4.11 Execution slowdown due to CTP controlled RDRAM using our predicted code layout and traditional layout. Both are compared to a static system with conventional DRAM. 8K unified cache is used. ... 28
List of Tables

3.1 Memory Power State and Transition Values: All accesses incur 90 ns access time plus possible resynchronization delay. We assume going from higher power mode to lower mode takes no time or energy. . . . 14

3.2 The break-even times for a power transition to become beneficial. For example, there must be at least 10 ns spent in standby mode before an Active->Stby transition really saves energy. . . . . . . . . . . . . . . . . . . 15

4.1 Time comparison between static analysis and simulation . . . . . . . . 19
Low power has become one of the standard design criteria of today’s battery-powered embedded or portable systems. Main memory takes a large portion of the whole system power budget as these systems usually employ an optimized low energy processor and do not have a display, hard disk or network devices. In [1], the memory system is shown to consume more than 90% of the total energy. As a result, reducing the energy cost of main memory is becoming increasingly important. Modern memory architectures, such as RDRAM, can be operated in several different power modes, with lower power modes employed when memory is not expected to be referenced. However, requests can not be serviced while in low power mode, and thus a resynchronization cost is incurred to bring memory back to active mode. In this thesis, we examine a partitioned memory architecture, consisting of multiple banks of memory. Each bank can independently transition between power modes.

In this thesis we show that traditional placement of program code, stack, and heap data does not take full advantage of multiple power modes. The usual layout is to have the code segment in the lower address space, global data and heap follow the code, while the stack resides in the high address range. A different layout, however, may result in different memory behavior, with the net result of lower the overall energy consumption of the program. One choice would be to place the code segment in the high address range, nearer the stack.

In a cacheless partitioned memory architecture, the memory bank with the code is typically referenced very frequently (and thus is always in active mode). Access
to any other data that is stored in that bank will never have to pay the resynchronization cost (in energy or time). In the traditional layout, it is the heap data that benefits, because it is located near the code segment. However, if a program makes more references to the stack than the heap, choosing an alternative layout will reduce the overall energy consumption. This is primarily due to the fact that the memory bank containing the heap will have more opportunities to transition to a low power mode (due to less frequent accesses). In our experiments, we observe that for some programs the memory energy reduction for this layout is significant (up to 43%).

We developed a static program analysis technique to estimate at link time how many references are made to the stack and heap, and based on that, to predict which layout (traditional or non-traditional) is more energy efficient. Our analysis is based on a relatively simple assessment of the control flow, but despite only limited information, we correctly predict 13 out of 15 benchmarks from the MiBench suite [2].

We measured the energy and performance impact of the different layouts through simulation on a modified Simplescalar based energy simulator (sim-panalyzer)[4]. We augmented the simulator to include a power aware memory architecture and a memory controller implementing the Constant Threshold Policy (CTP) proposed in[5]. Our results show that 11 out of 15 benchmark programs from the MiBench benchmark suite benefit from the modified layout. In a cacheless environment, the memory energy reduction over the traditional layout ranges from 2% to 43%, with an average of 12%. In addition, the resynchronization times are generally cut in half, though this has little effect on execution time. With 8K split cache or unified cache, energy reduction is about 1% with almost the same execution time, though some programs observe as much as a 25% reduction in energy from the non-traditional layout.
Chapter 2

Layout Prediction

We have developed a static program analysis technique to estimate the relative ratio of heap to stack accesses for a given program. If we predict the ratio to be lower than one (more stack accesses), we prefer to place the code in the high address range along with the stack. If the predicted ratio is greater than one, we prefer the traditional layout (code in the low address range). The implementation of the new program layout is trivial, and we do not fully describe it here. It involves a small change to the linker script, and some extra information stored in the executable.

The remainder of this section describes our static analysis, and includes a simple example of our analysis in operation.

2.1 Static Analysis

Our analysis is a simple static analysis which attempts to compute the total number of stack and heap references occurring in a program. Our analysis proceeds in three major steps. First, we build a basic block level control flow graph for each function. Next we do an inter-procedural analysis of stack registers(i.e. registers pointing to the stack). Based on the stack register information, we then compute the number of stack and heap references in each function invocation, ignoring function calls made within, and finally, we process the call graph to accumulate the references due to all function calls. We examine the computed values for the function main, which includes all functions called from main to predict the computed layout. If there are
more heap references, we predict the traditional layout, otherwise we predict the modified layout.

While a more powerful analysis may treat user code and library code equally, our analysis does distinguish between user code and library code with respect to both loop count estimation and recursion. We make the assumption that loop code in a user function is executed 30 times as often as the surrounding code (this means that a doubly nested loop is executed 900 times), while loop in a library routine is executed 6 times as often. We assume recursive functions reach a nesting depth of 10 in user code and 2 in library code. This means that if a user function directly or indirectly calls itself, our estimate is that there are 10 recursive invocations per initial invocation. This differentiation is due to the observation that the loops in a user function often have larger numbers of iterations (for example, benchmark program sha, rawaudio, rawaudio, etc) and there are more function invocations caused by a recursive call in user code. The actual numbers above are obtained through series of experiments of different constants. We picked the best performing numbers. Note that despite these simplistic assumptions, our predictor is quite accurate.

2.1.1 Control Flow and Block Weight Computation

Each basic block is examined for load and store instructions (these are the only memory operations on a RISC machine). We consider any reference through a stack register (the frame pointer (fp), the stack pointer (sp) or any register derived from them) to be stack references. We ignore all other instructions other than control flow instructions, which we use to construct a control flow graph.

First, we construct a basic block level control flow graph for each function, identifying loops and loop header blocks. After constructing the graph, we assume that all control flow paths out of a block are equally likely. When this algorithm is fin-
**Proc** Comp-Weight-Func

 push(Stack, entry)
 while (not done) do
   done := true
   while (t := pop(Stack)) do
     mark(t)
     forall x := succ(t) do
       x.weight += t.weight / numsucc(t)
       if all-preds-marked(x) then
         push(Stack, x)
       endif
     endfor
   enddo
   if $\exists$ x s.t. !all-preds-marked(x) then
     /* irreducible flow, pick one */
     push(Stack, x)
     done := false
   endif
 enddo
endProc

Figure 2.1: Pseudo-code for block weight computation

ished, all blocks in the function have been assigned a weight. The pseudo-code for our algorithm is shown in Figure 2.1.

Initially, we set the entry block for a function to be executed a single time, and all other blocks are set to be executed 0 times. We then flow weight values through the graph in the following way. If a block has $n$ successors, we assign $1/n$ of its weight to each successor. Blocks remain unprocessed until all predecessors have been processed. Loop headers will be processed when all non-loop predecessors have been processed. Loop headers are weighted at 30 times the weight of non-back edges entering the block for user code, and 6 times the weight for library code. We also ensure that the sum of the weights entering a loop is equal to the sum of the weights
leaving a loop. We iterate this process until all basic blocks receive weights from their predecessors.

When the above iteration halts, one of two conditions is true. Either all of the blocks in the function have been assigned weight, or there is irreducible control flow in the function. If all blocks have been assigned, we move on to the stack register analysis described below. If not, we arbitrarily break the dependency between one of the blocks causing the irreducible control flow and proceed as above.

2.1.2 Stack Register Set Computation

For each basic block we attempt to compute the set of registers that point to the stack (stack register set). Data processing and load/store instructions can both affect the set of stack aliases. For instance, both ‘mov r12, sp’ and ‘sub r12, sp, 30’ will cause r12 point at the stack. Store/load instruction pairs may also cause a register to point at the stack. For instance, one instruction may store the stack pointer through a register, and a subsequent instruction loads that same value into a second register. After the load instruction, the second register also points at the stack. We attempt to track registers through memory to handle such load/store pairs. The analysis of the load/store pairs is necessarily imprecise, because we cannot know in general if any two registers are aliases.

2.1.3 Intra-procedural Stack Register Analysis

During our intra-procedural analysis, we must handle procedure call instructions conservatively. Our analysis uses the stack register set from the exit block of the callee as the input set for the return block. The pseudo-code for stack register analysis of a basic block is shown in Figures 2.2 and 2.3.

We process each function as described above. The set of stack registers at function entry is determined by the call site. For the ‘main’ function, the initial set is \{sp,
fp}, the original stack and frame pointers. A function gets processed when all its call sites’ exit stack register sets are computed. The union of those exit sets is set as the entry set of the new block.

**Proc** Compute-StackRegSetBlk(bblk)

StackRegSet := bblk.entryStackRegSet

forall insn in block do

   switch insns.op {

      case LOAD:

         if source(insn) is stacklocation then
            StackRegSet += dest-reg(insn)
         else
            StackRegSet -= dest-reg(insn)
       endif

      case STORE:

         if src-reg(insn) in StackRegSet then
            destination(insn) is stacklocation
         endif

      case MOVE:

         if src-reg(insn) in StackRegSet then
            StackRegSet += dest-reg(insn)
         else
            StackRegSet -= dest-reg(insn)
       endif

      case BINARY-OP: /* add, sub, etc. */

         if src1(insn) in StackRegSet and
            src2(insn) is immediate then
            StackRegSet += dest-reg(insn)
         else
            StackRegSet -= dest-reg(insn)
       endif

   endswitch

endfor

bblk.exitStackRegSet := StackRegSet

**endProc**

Figure 2.2: Pseudo-code for stack register analysis in a block
2.1.4 Inter-procedural Stack Register Analysis

The whole program stack register analysis starts from `main`. We process each function invocation in the sequence that occurs in the code, maintaining a call stack. A function invocation is first pushed to the call stack when it gets processed. We then do an intra-procedural stack register analysis (described above) for the current function invocation. During this, if we encounter a function call not in the current call stack, we’ll push this function onto the stack and process the new function with the current stack register set as its entry set. When a callee is finished, it is popped out of the call stack and the exit stack register set is handed to the caller function and the caller function can continue to process itself. If the target function is already in the stack, we simply mark it as recursive and skip over the new call.
2.2 Memory Reference Estimation

The final estimation process starts from the ‘main’ function. For each function invocation, we first do the stack register analysis described above. Then we compute the base memory references occurred within the function itself (without considering any callee’s). Later, the callee’s are processed and their memory references are added to the caller. Note that when we incorporate a callee’s references to the caller, the number of references is scaled by the weight of the block making that call. Pseudo code for our memory reference estimation is shown in Figure 2.4

```
Proc Compute-Ref-Func(func,entryStackRegSet)
    push(callStack, func)
    Compute-StackRegSetFunc(func,entryStackRegSet)
    Compute-Base-Ref-Func(func)
    forall callee of func do
        if (callee in callStack) then
            isRecursive := true
        else
            tempStackRegSet := call-block.StackRegSet
            if ! Processed(callee, tempStackRegSet) then
                Compute-Ref-Func(callee, tempStackRegSet)
            endif
            func.stackRef += callee.stackRef*blockweight
            func.heapRef += callee.heapRef*blockweight
        endif
    endfor
    if isRecursive == true then
        func.stackRef *= recursiveWeight
        func.heapRef *= recursiveWeight
    endif
    pop(callStack)
endProc
```

Figure 2.4: Pseudo-code for memory reference computation (Note loopWeight and recursiveWeight can be different between user function and library function)

Estimating the number of stack and heap references depends on the stack register analysis. If we assume that only the stack and frame pointers refer to the stack,
we will get an inaccurate picture of the ratio of stack and heap references. In our experiments, only 3 out of 15 predictions are correct under this assumption.

2.3 Prediction

After processing the entire control flow graph of the program, we must make a prediction as to which layout is more energy efficient. If we predict that the ratio of heap to stack is greater than one, we predict the traditional code layout. If the ratio is less than one, we predict the modified layout.

2.4 Example

Consider the C program shown in Figure 2.5. We first construct the corresponding control flow graph (including ARM assembly code) shown in Figure 2.6.

```c
int Global=5;

void foo(int *p)
{
    *p = 1;
}

void main()
{
    int local;
    for (local = 0; local <= Global; local++)
    {
        foo(&local);
    }
}
```

Figure 2.5: Simple example C code
Figure 2.6: Control flow graph for the code shown in Figure 2.5. Loop blocks are shown in gray, stack instructions are marked with an S, and heap instructions are marked with an H. Instructions marked with C accesses code memory, which contains a pointer to the static data in this case.

Loop blocks (weight 30) are shown in gray, other blocks are shown in white. Instructions which reference the stack are marked with an S, while instructions which reference the heap are marked with an H. The mark C means the instruction references the code memory. An example is `ldr r1, [pc,#34]`, where [pc,#24] is a pointer to a global static data object( ) . We ignore this kind of memory reference because they are just like instruction references. In each block, the entry and exit stack register set is shown at the beginning and the end. The registers in the parentheses next to an instruction are the stack register set before the execution of that instruction. Note the stack register set is not always accurate due to the fundamental difficulties of the register aliasing problem.
‘ldm’ is a bulk load, which can load multiple consecutive words in memory to a list of registers. For each word, there is one memory access. Similarly, ‘stm’ is a bulk store, which stores the contents of a list of registers in a consecutive memory region. For example, ‘stmdb sp!, \{r11,r12,lr,pc\}’ in block 0 of function ‘foo’ is a bulk store accounting for 4 stack references.

Initially r11 and r13 are the only stack registers. They are the entry set for block 0 of ‘main’. After the first instruction ‘mov r12,sp’, r12 is added to the stack register set. r0 is added after ‘sub r0, r11, #20’ in block 1. Then the inter-procedural stack register aliasing takes place at the call site in block 1. The call site stack register set becomes the entry set of foo. In foo, ‘str r3, [r0]’ is correctly predicted to be a stack reference.

The function main itself has 40 (=10+2*30*0.5) stack references, and 16 (=1+1*30*0.5) heap references. Similarly, the function foo has 8 stack references and 0 heap reference. Because main calls function foo the counts from foo must be added to the counts for main. foo is called from a loop block, so its counts are scaled by 30 before being added to the counts for main. Therefore, the total stack references for main is 160 (=8*30*0.5 + 40), and the total heap references is 16. Because our analysis determines that there are more stack references than heap references, we predict that the non-traditional code layout is better.
Chapter 3

Experimental Architectures

We consider several simple, though we believe typical, architectures for low-end embedded or portable systems. In these designs, memory energy represents a significant overhead, and we expect to see more partitioned memory subsystems to reduce the power consumption of these systems. Program layout can affect such systems significantly. Our architectures are all based on actual ARM processors, with varying memory and cache configurations. The three main features of our setup include the processor, the multi-bank memory subsystem, and our modified simulator, which we describe in turn below.

3.1 Processors

We consider three different ARM-based [8] configurations. The ARM7 core is a 32-bit embedded RISC processor optimized to provide the best combination of performance, power, and area characteristics. It is a widely used processor in wireless communications [9], multimedia applications, digital cameras, PDA’s, etc. One important feature of the ARM7 processor is that it has no MMU, so virtual memory common to today’s consumer PC architecture does not exist. Instead, programs directly reference physical addresses. Because we were interested in exploring the our technique on different memory hierarchies, we examined three different memory configurations – no cache (ARM7TDMI), 8 Kb unified cache (ARM720T), or split 4k instruction and 4k data caches (ARM940T). In our experiments, we simulated
Table 3.1: Memory Power State and Transition Values: All accesses incur 90ns access time plus possible resynchronization delay. We assume going from higher power mode to lower mode takes no time or energy.

<table>
<thead>
<tr>
<th>Power state or transition</th>
<th>Power(mW)</th>
<th>Time(ns)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Active</td>
<td>300</td>
<td>$t_{access}=90$</td>
</tr>
<tr>
<td>Standby</td>
<td>180</td>
<td>n/a</td>
</tr>
<tr>
<td>Nap</td>
<td>30</td>
<td>n/a</td>
</tr>
<tr>
<td>Powerdown</td>
<td>3</td>
<td>n/a</td>
</tr>
<tr>
<td>Stby→Act</td>
<td>240</td>
<td>$D_{s→a}=5$</td>
</tr>
<tr>
<td>Nap→Act</td>
<td>165</td>
<td>$D_{n→a}=60$</td>
</tr>
<tr>
<td>Pdn→Act</td>
<td>152</td>
<td>$D_{p→a}=25,000$</td>
</tr>
</tbody>
</table>

3.2 Memory Subsystem

We used the same partitioned power aware memory system proposed in [10]. There are 4 power modes: active, standby, nap and power-down. Note that even in power-down mode the memory contents are still retained. Power consumption becomes less when a bank is transitioned into lower power modes, but a resynchronization is needed to put it back to active when a memory request is serviced. The power, memory access time and resynchronization cost are shown in Table 3.1.

Placing memory bank in lower power modes can or cannot save energy depending on how much time is spent in a low power mode. We can calculate this break-even time by the following formula,
\[ \text{break-even time} = \frac{\text{transition power} \times \text{resync time}}{\text{high power} - \text{low power}} \] (3.1)

The break-even times are shown in Table 3.2. We only present values for transitions from active to low power modes. The break-even times for an transition between low power modes are not calculated because such transitions do not typically occur.

<table>
<thead>
<tr>
<th>Power transition</th>
<th>Break-even Time(ns)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Active-&gt;Stby</td>
<td>10</td>
</tr>
<tr>
<td>Active-&gt;Nap</td>
<td>66</td>
</tr>
<tr>
<td>Active-&gt;Pdn</td>
<td>140,740</td>
</tr>
</tbody>
</table>

Table 3.2: The break-even times for a power transition to become beneficial. For example, there must be at least 10 ns spent in standby mode before an Active->Stby transition really saves energy.

We assume there is a smart memory controller implementing the Constant Threshold Policy (CTP) proposed in [5]. This policy directs the memory bank into next lower power mode after some predefined threshold time of inactivity. In our case, we transition the memory from active to standby after 20 cycles of inactivity, from standby to nap after an additional 200 cycles, and from nap to power-down after an additional 100,000 cycles. Those numbers are only empirical. In some cases, the CTP policy can cause the RDRAM based systems to use more energy than a static system. If a memory bank transitions to standby mode immediately before a reference, the processor has to wait (using energy) while the memory resynchronizes.

### 3.3 Simulator

**sim-panalyzer** is a Simplescalar based simulator, designed to model the ARM processor. It is verified against MARS, a Verilog ARM core implementation[3]. We augmented **sim-panalyzer** with multi-bank memory and a CTP memory controller. Energy consumption is computed by accumulating the power values of all memory banks (possibly in different power modes) at every cycle. Because the memory bank
configuration can have a significant effect on the power consumption we considered 3 different configurations for 2 MB of memory. We used 2 banks of 1MB each, 4 banks of 0.5MB each, and 8 banks of 0.25MB, with each bank independently controlled.

Memory in the simulator uses sequential bank addressing. That is, each bank represents a contiguous block of memory. An alternative approach is to use bank interleaving. This approach spreads contiguous memory addresses across all memory banks. In traditional architectures this approach is used to increase the parallelism exposed to the memory system. However, as shown in [12], sequential bank addressing is more energy efficient. In addition, because current RDRAM supports internal bank interleaving, external interleaving does not improve performance significantly.
Chapter 4
Experimental Results

We evaluated the effectiveness of our static analysis by running programs from the MiBench benchmark suite on an augmented version of sim-panalyzer as described in the previous section. The MiBench suite is a set of typical embedded applications. We were not able to run all of the benchmark programs in our setup, we report in this section the results of all programs which executed correctly on the simulator. All benchmarks were statically compiled with a standard gcc cross compiler at optimization level 3(-O3).

We consider only the energy consumed by the application program. In most embedded and portable systems, the operating system is typically either very rudimentary and small([13]) or does not exist at all.

In our experiments we used 15 benchmark programs. We measured the code size, as well as the maximum stack and heap sizes. None of the code segments is larger than 260KB. The stack and heap sizes vary from about 10KB to 1MB with different programs.

4.1 Static Prediction Evaluation

We also compared our prediction of the heap/stack reference ratio with the real ratios measured in simulation. We present the results in Figure 4.1. If the predicted layout is less than one, our algorithm selects the modified layout. If the predicted ratio is greater than one, our algorithm selects the traditional layout. Simulated shows
Figure 4.1: heap/stack reference ratios. We predict correctly if both ratios are above or below 1 at the same time.

The actual ratio measured in simulation. We count rawaudio and stringsearch as mis-predictions. For rawaudio, both simulated and predicted ratios are very close to 1. The simulated ratio is a little above 1 while the predicted ratio is a little below 1. We take it as below 1 and predict the modified layout. The simulated ratio of stringsearch is also very near 1. However, the predicted ratio is well above 1. We predict the traditional layout in this case. sha is dominated by stack references, so the ratio is almost 0.

Our analysis takes about 2.6 to 5 seconds to complete a benchmark program while the simulation of the same program can take hundreds of times longer. The following table (4.1 shows the actual time spent in the analysis and simulation (done with no cache configuration) on a PC with a 2GHz Intel Xeon processor and 1G RAM.

Our analysis does have some additional limitations. First, we ignore indirect function calls (due to the use of function pointers). Our call graph is therefore not precise,
<table>
<thead>
<tr>
<th>Program</th>
<th>Static Analysis Time(second)</th>
<th>Simulation Time(second)</th>
</tr>
</thead>
<tbody>
<tr>
<td>bitcnts</td>
<td>4.446</td>
<td>2126</td>
</tr>
<tr>
<td>blowfish</td>
<td>2.673</td>
<td>4138</td>
</tr>
<tr>
<td>cjpeg</td>
<td>4.765</td>
<td>1557</td>
</tr>
<tr>
<td>crc</td>
<td>2.695</td>
<td>5262</td>
</tr>
<tr>
<td>dijkstra</td>
<td>4.089</td>
<td>3669</td>
</tr>
<tr>
<td>djpeg</td>
<td>5.013</td>
<td>443</td>
</tr>
<tr>
<td>fft</td>
<td>2.793</td>
<td>3171</td>
</tr>
<tr>
<td>ispell</td>
<td>3.072</td>
<td>560</td>
</tr>
<tr>
<td>patricia</td>
<td>4.268</td>
<td>5993</td>
</tr>
<tr>
<td>rawcaudio</td>
<td>2.592</td>
<td>1305</td>
</tr>
<tr>
<td>rawdaudio</td>
<td>2.655</td>
<td>1190</td>
</tr>
<tr>
<td>rijndael</td>
<td>2.848</td>
<td>2118</td>
</tr>
<tr>
<td>sha</td>
<td>2.656</td>
<td>604</td>
</tr>
<tr>
<td>stringsearch</td>
<td>2.631</td>
<td>7</td>
</tr>
<tr>
<td>untoast</td>
<td>2.994</td>
<td>663</td>
</tr>
</tbody>
</table>

Table 4.1: Time comparison between static analysis and simulation

because we do not determine the destination of some calls. Examination of most of our benchmark program indicates that indirect calls take up only a very minor portion of all function calls, and they lie primarily in library code. The exceptions are cjpeg and djpeg, which make extensive use of indirect function calls. However, the predicted ratio still matches well with the simulated ratios. This limitation could be ameliorated using techniques presented in [14].

4.2 Stack Register Analysis Evaluation

Stack register analysis is an important step to predict whether a memory instruction accesses heap or stack. We collected all the predicted heap/stack instructions and checked them with the results from the simulation. Figure 4.2 and Figure 4.3 show the percentage of predicted heap/stack instructions that indeed accesses heap/stack in
the simulation. Note the instructions not executed in the simulation are not counted. On average, the accuracy is about 80-85%. However, \texttt{fft} and \texttt{blowfish} do not perform well on heap instruction prediction. For \texttt{blowfish}, the reason is we are unable to recognize a double pointer to the stack. For example, in ‘\texttt{foo(char* argv[])}’, we probably can know if \texttt{argv} is on stack or heap, but we do not know anything about \texttt{argv[0]} in our analysis. Since our analysis assumes everything undecided to be an heap instruction, a mistake will be made if it turns out to be a stack instruction. For \texttt{fft}, the problem lies in a number of library functions, on which our analysis does not do well. If we can profile the library functions beforehand, the results should be better.

![Heap Instruction Prediction Accuracy](image)

Figure 4.2: The percentage of predicted heap instructions that are indeed heap instructions in the simulation.

4.3 Energy Savings

Figure 4.4 shows the energy savings of traditional layout in a cache-less system by using CTP-controlled RDRAM instead of conventional DRAM. \texttt{2_1_mem} shows the energy savings when considering only the memory, using 2 1MB independent memory
Figure 4.3: The percentage of predicted stack instructions that are indeed stack instructions in the simulation.

banks. $2_{-1} total$ shows the energy savings when considering the entire system (processor and memory). Likewise, $4_{-0.5} mem$, $4_{-0.5} total$, $8_{-0.25} mem$ and $8_{-0.25} total$ show the memory and total system energy savings using 4 0.5 MB and 8 0.25 MB independent banks respectively. Generally, the base total system energy savings on cache-less processor are around 47% over all 3 memory configurations. Some programs like untoast achieve significantly more energy savings with 8-bank configuration. This results from a side effect of our program layout change. In the traditional layout, the code and heap of untoast are large enough to occupy 2 banks, so that 3 banks (the other is stack) are used in total. However, in the modified layout, the code and stack fit in 1 bank, which means only 2 banks acting. Since fewer banks are used in the modified layout, energy consumption is lower.

Our scheme saves an additional 12% energy on a cacheless processor. Figure 4.5 shows the energy savings obtained by using our predicted layout compared to the
Figure 4.4: Energy savings due to CTP controlled RDRAM using the traditional layout, compared to a static system with conventional DRAM, which is always in high power mode. No cache is involved.

traditional layout in such a system. The savings from choosing the correct layout can be as much as 40% of the total system energy. **sha** (a secure hashing algorithm), and **fft** (fourier transform) use the stack almost exclusively and thus benefit the most from the non-traditional layout. In some cases the non-traditional layout uses fewer memory banks than the traditional layout. This results in energy savings, because more of the banks can be placed in low power mode. Examples include **blowfish**, **ispell**, **rijndael** and **untoast**.

There are 4 programs (**dijkstra**, **djpeg**, **rawaudio** and **stringsearch**) predicted to use less energy in the traditional layout. In Figure 4.6, we show about 15% energy increase in the modified layout over the traditional. In 8-bank configuration, **dijkstra** uses fewer banks in the modified layout and achieves energy saving.
Energy Savings

Figure 4.5: Energy savings of our predicted code layout over traditional layout with no cache

in fact. jpeg uses 3 banks in both layouts and the energy consumption is roughly equal.

Figure 4.7 shows the energy savings for the different memory configurations when using split instruction and data caches of 4 KB each. On average, 1% energy saving is achieved and execution time is nearly the same. The advantage of our scheme is smaller than in a cacheless setting because many of the original memory references are filtered by the cache. There are still some programs (such as sha) which save energy using our predicted layout. The one benchmark which performs poorly is rijndael, which has poor cache performance. In this program, both the code and heap exhibit poor locality, but the stack exhibits good locality. Because our prediction indicates that rijndael contains more stack than heap references, we place...
the code with the stack. However, the poor locality of the code references causes the stack/code bank to be in high power more more often using the non-traditional layout. In most other cases, our predicted layout and the traditional layout are nearly identical.

Figure 4.8 shows the energy savings for the different memory configurations when using a unified 8K cache. The energy savings is comparable to the split cache configuration. Interestingly, sha does not get much improvement as before. The reason is that sha has small code, which means it makes poor use of the 4KB instruction cache in the split cache configuration. In contrast, the blowfish benchmark improves substantially in the non-traditional layout. In this case, the code, which had good cache performance under the split cache configuration, conflicts with the data (both stack
Energy Savings

![Energy Savings](image)

Figure 4.7: Energy savings of our predicted code layout over traditional layout with 4K I-cache and 4K D-cache

and heap) causing more cache misses. The non-traditional layout therefore results in fewer resynchronizations and more time in low power mode.

### 4.4 Execution Time

In addition to energy savings, we must consider the impact of our scheme on execution time. Figure 4.9,4.10 and 4.11 show the execution slowdown of our predicted layout compared to the traditional layout with different cache and memory configurations. For some programs we predict that the traditional layout is the correct one (and thus have identical slowdown), and for others we choose the non-traditional layout. In most cases, the execution time does not change significantly, despite the energy savings. Thus we are able to improve the energy consumption of several pro-
Figure 4.8: Energy savings of our predicted code layout over traditional layout with 8K unified cache.

grams without sacrificing performance. In those figures, 4-bank configuration looks very similar to 2-bank configuration because programs still use 2 banks, in fact. However, with 8 banks, some programs are faster in modified layout. This is because fewer banks are used as described above, and thus fewer resynchronizations are incurred.
Figure 4.9: Execution slowdown due to CTP controlled RDRAM using our predicted code layout and traditional layout. Both are compared to a static system with conventional DRAM. No cache is used.

Figure 4.10: Execution slowdown due to CTP controlled RDRAM using our predicted code layout and traditional layout. Both are compared to a static system with conventional DRAM. 4K I-cache and 4K D-cache are used.
Figure 4.11: Execution slowdown due to CTP controlled RDRAM using our predicted code layout and traditional layout. Both are compared to a static system with conventional DRAM. 8K unified cache is used.
Chapter 5

Related work

There exists a fair amount of research targeting multi-bank memory architectures at different levels. There seem to be two general research approaches. One is to predict the memory inter-access time so that memory bank can be effectively turned to lower power mode for a while. The other is to cluster objects of high time affinity together in the same bank. The objects can be data objects at the compiler level, or pages and processes at the operating system level. We describe the existing research in detail below.

At the hardware level, Delaluz et. al.[5], propose a set of threshold predictors built into a memory controller to manage the power mode transition. They are adaptive threshold predictor (ATP), constant threshold predictor (CTP), and history-based predictor (HBP). The rationale behind these predictors is that if a memory bank has not been accessed in a while, then it is not likely to be needed in the near future, and on the other hand, it is likely to be accessed again soon if it was just accessed. A threshold is used to determine the idleness of a bank after which it is transitioned to a lower energy mode. In ATP, the threshold is adaptive. It starts with an initial threshold. If the next access is soon after the first access, the resynchronization cost is relatively high compared to the savings from the mode transition, so the threshold gets doubled. If the next access is fairly late, it means the threshold is too long and it is set to the initial value. CTP uses constant thresholds for mode changes. We use the CTP in our experiments. HBP predicts the next inter-access time by
inspecting the history of previous accesses and set the threshold accordingly. Fan et al. [11] developed an analytical model that approximates the idle time of memory bank using an exponential distribution for cache-based systems. Their results reveal that, for most workloads on cache-based systems, memory banks should immediately transition to a lower power mode when they become idle and will not benefit from more sophisticated power management policies.

At the compiler level, Delaluz, Kandemir, et al. [1], applies several traditional compiler techniques (e.g. array allocation, loop and data transformation) to help clustering the data objects of high temporal affinity in the same bank. In [6], Delaluz et al. also proposes a special array allocation optimization technique, which tries to interleave large array spanning multiple banks so that simultaneously accessed array elements are put in the same bank. The limitation of their work is that their techniques only apply to loop and array dominated programs. The effects of their techniques on more general programs is not clear. They also, in [7], proposes strategies to automatically place arrays with temporal affinity into the same set of banks at run time.

At the OS level, Lebeck et al. [12], evaluates energy effects of page allocation schemes complemented with dynamic memory controller policies. Their major finding is that the sequential-first-touch page allocation achieves more energy saving than the random page allocation in a conventional operating system. They also experiment with a frequency-based clustering page allocation scheme. However, it is not shown to save more energy than the sequential one. In [10], Fan et al. takes CPU dynamic voltage scaling (DVS) into consideration. This is a more systematic approach because DVS is a common method to save CPU energy and changing CPU frequency can affect inter-access time (which is critical to the threshold-based hardware policy) observed by the memory subsystem. Delaluz et al. [15], implements an energy-aware process scheduler. Their idea is to put processes using the same
set of banks in the same scheduling slot. In [16], Huang et. al. implements a power aware virtual memory system. Their system keeps track of the memory banks used by each process and has a variety of techniques to minimize the number of active banks.

Whole program optimization has been studied with respect to improving performance. In *Spike*[17], *OM*[20], *alto*[21] and other link-time optimizers, a number of whole program code improvements are performed. These include code layout to improve instruction cache behavior [19], hot cold optimization [18], register allocation, inter-procedural code motion, etc. These approaches typically do not consider heap or stack data, and are considerably more complex. Our approach bears some similarity to link-time optimizers such as these, though the goals are different. We wish to make a simple decision, and require a much less sophisticated algorithm. Despite a less sophisticated analysis, we are able to make quite accurate predictions. Furthermore, the above full-fledged optimizers are known to be slow in operation due to its complex analysis. Our system, in contrast, is quite fast as shown in table

Program profiling is a tool to enhance optimizing compilers and linkers. More optimization opportunities can be revealed by a program’s run-time information. Some optimization techniques mentioned above, such as code layout and hot cold optimization, rely on profiling information. Other profile-driven optimizations include storage allocation [26], code compression [22], instruction mixing [25], value-based optimizations [23][24](e.g. constant propagation, code specialization), etc. Despite its wide applicability, profiling has some inherent limitations[27]. For example, a representative set of inputs is required, which is hard to validate. In addition, profiling can be very time consuming. Our system uses static analysis only and thus avoids the drawbacks of profiling.

Researchers also perform whole program optimizations at the compiler level. Vortex [28] targets a class of object-oriented languages, and *Fortran-D* [29] targets
parallel and distributed scientific programs. They are more application-aware and language-specific. Our technique is applied at the linker/loader level, and is applicable to any high level language. Compile-time and link/load-time optimizations are fairly orthogonal and complementary to each other. While link/load-time optimizations are less language dependent, compile-time optimizations are less machine dependent.
Power and energy management is a crucial enabling technology for future high-performance handheld computers and embedded devices. Especially, main memory can take as high as 90% of the whole system power budget in these systems[1]. As a result, reducing the energy cost of main memory is becoming increasingly important. Modern memory architectures, such as RDRAM, can be operated in several different power modes, with lower power modes employed when memory is not expected to be referenced. However, requests can not be serviced while in low power mode, and thus a resynchronization cost is incurred to bring memory back to active mode.

In this thesis, we explored a partitioned memory architecture, consisting of multiple banks of memory. Each bank can independently transition between power modes. Applications typically directly access physical memory in such systems.

We investigated a new placement of program code, which is putting code and stack together, instead of code and heap traditionally. In a cacheless partitioned memory architecture, the memory bank with the code is typically referenced very frequently (and thus is always in active mode). Access to any other data that is stored in that bank will never have to pay the resynchronization cost (in energy or time). In the traditional layout, it is the heap data that benefits, because it is located near the code segment. However, if a program makes more references to the stack than the heap, choosing an alternative layout will reduce the overall energy consumption. This is primarily due to the fact that the memory bank containing
the heap will have more opportunities to transition to a low power mode (due to less frequent accesses). Through our experiments, we found out that the new layout indeed results in up to 43% lower memory and overall energy consumption of some MiBench programs over traditional layout in a cacheless environment.

We developed a static program analysis technique to predict which layout is more energy efficient. The technique basically estimates how many references are made to stack and heap at link time. If there are more stack references, the new layout is chosen. Our analysis is based on a relatively simple assessment of the control flow, but despite only limited information, we correctly predict 13 out of 15 benchmarks from the MiBench suite [2].

We also evaluated our scheme on a simulated system with cache. Although in general, the energy saving is not significant, it is as high as 25% for some benchmark programs. For all the configurations (including cacheless) we have experimented with, we see little execution slowdown.
Bibliography


