A Trace-based Binary Compilation Framework for Energy-Aware Computing

Lian Li and Jingling Xue
Compiler Research Group
School of Computer Science and Engineering
University of New South Wales
Sydney, NSW 2052, Australia
National ICT Australia

{lianli,jxue}@cse.unsw.edu.au

ABSTRACT

Energy-aware compilers are becoming increasingly important for embedded systems due to the need to meet conflicting constraints on time, code size and power consumption. We introduce a trace-based, offline compiler framework on binaries and demonstrate its benefits in supporting energy optimisations. The key innovation lies in identifying frequently executed paths in a binary program and duplicating them as single-entry traces. Separating frequently from infrequently executed paths enables the compiler to focus both performance and energy optimisations on the hot traces.

Traces constructed at the level of binaries are inherently inter-procedural, spanning both application and library code. Such a framework allows an embedded application developer to exploit optimisation opportunities made possible due to the information that is available only at link time.

We describe the implementation of our trace-based framework in `alto`, a link-time optimiser for the Alpha architecture. We present a new algorithm for constructing the hot traces from binaries. This algorithm is both effective (since the execution cycles are mostly spent on traces) and practical (due to small code size increases caused). We have developed and implemented a new optimisation to reduce the functional unit leakage energy. We show how the traces facilitate the development of such an optimisation, which results in significant leakage energy savings for benchmark programs at the cost of small performance penalties.

Categories and Subject Descriptors

D3.4 [**Programming Languages**]: Processors—*Compilers*

General Terms

Algorithms, Languages, Experimentation, Performance

Keywords

Energy optimisation, trace, link-time optimisation, profile-guided optimisation, binary translation

1. INTRODUCTION

Power consumption is becoming a major concern in today's microprocessor industry. According to scaling theory [2], power delivery and dissipation will be the primary limiters to designing future microprocessors in deep submicron technologies. The widespread use of battery-driven embedded devices and the continuing quest for more powerful, portable chipsets to cope with increasingly more complex applications have also exacerbated the power problem.

Researchers have begun to address the energy consumption issue at all phases of system design, from circuit technology, micro-architecture design to software solutions. Over the years, significant progress has been made in the area of low-power circuit and system design [5]. But the software work has not enjoyed a similar level of success. One of the areas in much need of work is energy-aware compilers.

There has been a great deal of work on applying compiler optimisations to reduce power and energy consumption [12, 14, 19, 20, 26]. Most of these works investigate the effects of individual optimisations on power and energy usage. There has been relatively little work in the design and implementation of energy-aware compilers [10, 13, 24, 27].

A critical issue to develop an energy-aware compiler is to maximise both performance and energy benefits subject to some performance and energy constraints. Given that many embedded application programmers are willing to spend time tuning a program, we should provide tools for the programmer to exploit the optimisation opportunities that manifest themselves at various levels of abstractions, from the source code (if it is available) to its binary code.

Therefore, we envision a static (i.e., offline) binary translation framework as sketched in Figure 1 for embedded systems. A binary file, i.e., machine executable code file is read in, and an intermediate representation (IR) is constructed. A suite of analyses and optimisations are then applied to the IR subject to some performance and resource constraints. Guided by some execution profiling information of the program, the hot traces, i.e., the frequently executed paths across all functions in the application code and libraries are identified. These hot traces are then optimised

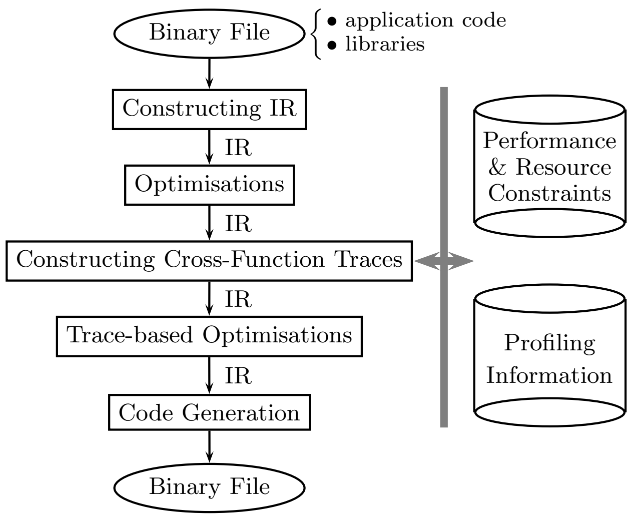

Figure 1: High-level structure of a static binary compilation framework for embedded systems.

subject again to the performance and resource constraints. Finally, the binary code is generated from the optimised IR.

We believe that the optimisations applied in such a binary translation system are complementary to those carried out by a compiler for the program initially written in a high-level language. The compiler can significantly improve the performance and energy usage of a program. However, the compiler typically compiles functions individually, and in addition, the inter-procedural optimisations that most compilers support are often limited in their extent and scope. Moreover, the compiler cannot optimise library calls at compile time when it has no access to the source code for the libraries. The problem is further aggravated for two reasons. First, some library routines in embedded applications often contain hand-written assembly code, which cannot be processed by the compiler. Second, embedded applications may include increasingly more components written in multiple languages with reusability in mind. As a result, the source code for some third-party libraries may not be available, and the hot traces can spread across the functions in both application and library code. All these problems can be addressed by carrying out link-time (or post-link-time) optimisations on binaries, by propagating information available only at link time across all the functions. Previous work on binary translation has demonstrated performance benefits even on highly optimised code [1, 6, 18]. This work demonstrates its benefits in reducing energy consumption.

In this paper, we report our preliminary experience on designing and implementing an instance of Figure 1 on top of `alto`, a link-time optimiser for the Compaq Alpha architecture [18]. The key innovation of this work lies in constructing and adopting hot traces as compilation units to support both performance and energy optimisations on binaries for embedded systems. Many recent techniques focus on adapting the configuration of processor resources in consonance with the changing application requirements [12, 19, 26]. In this direction, we see three important advantages with a trace-based binary translation framework:

Traces are the hot spots across the whole program. By performing inter-procedural analysis on binaries, the traces can represent quite accurately the hot spots across the procedural boundaries in both application and library code. The traces are flexible in accommodating a variety of control flow structures.

Traces can help the compiler explore energy-efficient architectural features to make tradeoffs between performance and energy consumption when required. The emphasis on processors with low power and high performance has resulted in the incorporation of energy-efficient hardware features into processor designs, such as dynamic voltage and frequency scaling (DVS) and power-aware instructions for turning on/off hardware components. We observe that the compiler techniques for exploiting these hardware features such as DVS [20, 25] and power-aware instructions for turning on/off cache lines [26] and functional units [19] share the following common pattern. The frequent switching on/off or voltage scaling activities can consume both CPU cycles and significant dynamic energy. To achieve power savings without jeopardising performance, the code regions in which the energy consumption is optimised (called *idle regions*) along with the suitable program points for inserting power-aware instructions must be judiciously identified. Traces facilitate these choices: traces can be used to form idle regions and the entries to and exits from these idle regions are candidates for insertion points.

Traces facilitate simultaneous optimisations on both performance and energy. An energy-aware compiler should manage the interactions between traditional performance-oriented optimisations and energy-oriented optimisations. How this can be done is still a research topic. Traces represent the hot spots where the most execution time and energy are spent. We believe that a trace-based framework will allow performance and energy optimisations to be carried out in concert by favouring the frequently executed paths while penalising the insignificant ones (if necessary).

In summary, the contributions of this paper include:

- a trace-based binary translation framework that is well suited for both performance and energy optimisations,
- an implementation of such a framework in `alto` for the Compaq Alpha architecture,
- a simple yet effective algorithm for separating frequently from infrequently executed paths in a binary program and duplicating the frequently executed paths as hot traces,
- a new algorithm for reducing the functional unit leakage energy in our trace-based framework, and
- experimental results demonstrating the benefits of our framework in supporting energy-oriented optimisations.

The rest of this paper is organised as follows. Section 2 introduces our trace-based methodology. Section 3 describes our algorithm for identifying and constructing the traces in a program. Section 4 describes our algorithm for reducing the functional unit leakage energy consumption in our

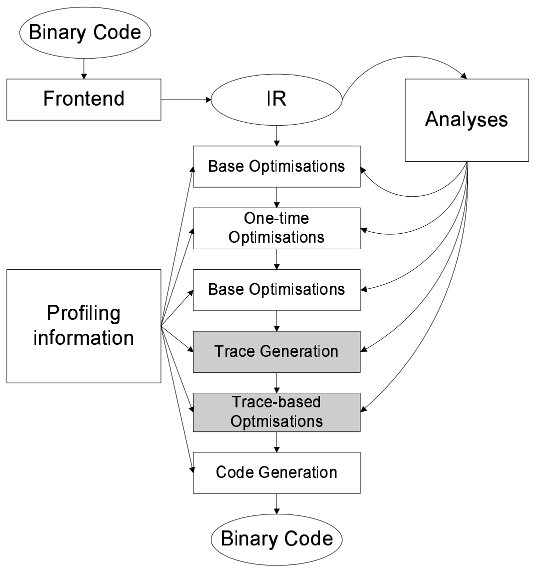

Figure 2: A traced-based binary compilation framework implemented in `alto` for the Alpha architecture.

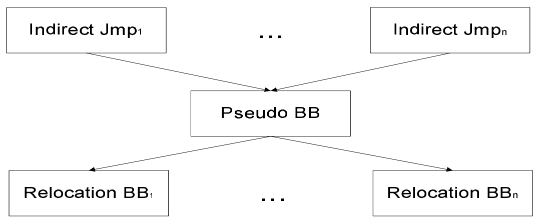

Figure 3: Unknown indirect jumps.

trace-based framework. Section 5 evaluates this work using benchmarks. Section 6 reviews the related work. Section 7 concludes the paper by discussing some future work.

2. TRACE-BASED METHODOLOGY

Figure 2 depicts the high-level structure of our trace-based framework for energy-aware computing on binaries. We have implemented our framework in `alto`, a link-time optimiser for the Alpha architecture [18]. So far we have added the two components indicated by the two boxes in gray.

Initially, a binary file is read in, and one single inter-procedural CFG (as the IR) is constructed from the binary code. At the level of binaries, it is sometimes difficult to extract precisely the control flow information about the program. For example, the targets of an indirect jump may not be resolvable. In this case, `alto` resorts to the relocation information from the linker. One single pseudo block is created. A control flow edge is added from every unknown indirect jump to the pseudo block, which has an outgoing edge to every block that can be potentially the target of the jump. The pseudo block serves merely as a boundary for all optimisations and is ignored in code generation.

Once the CFG is built, `alto` performs a suite of so-called base optimisations iteratively on the CFG. These include constant propagation, copy propagation, move elimination and value-based code specialisation. Then some one-time optimisations such as inlining and stack merging are applied only once to the CFG (to avoid undesirable side effects). To benefit from the opportunities created by the one-time optimisations, the base optimisations are repeated.

The "analyses" component is responsible primarily for performing data dependence analysis and control flow structure analysis (e.g., finding a loop and its header).

In `alto`, execution profiles are gathered by code instrumentation. The profiling information has been used to guide inlining, value-based code specialisation and code layout.

Once all the `alto` optimisations have been applied, our two components (in gray) will be in action. In Section 3, we describe our trace generation algorithm. In Section 4, we introduce our approach to reducing static power dissipation by functional units in our trace-based framework.

The code generation module carries out profile-guided code layout, code alignment and instruction scheduling.

Our framework supports only static binary optimisations, which are intended to be complementary to those carried out by the compiler. Applications that use shared or run-time libraries cannot be handled. In addition, static binary translators such as `alto` [18] rely on the reallocation information from the linker to reconstruct a CFG from binaries. This requires that all relocatable addresses be identifiable.

3. STATIC TRACE GENERATION

Based on edge profiling information, the frequently executed paths in the CFG are identified and duplicated as single-entry traces with the required control flow edges duplicated accordingly. This approach allows energy-oriented optimisations to be applied to the duplicated traces only. As we shall see in Section 5, the execution time of a program is mostly spent on the hot traces, which result in relatively a small increase in code size.

DEFINITION 1. *A* **trace** *is a sequence of basic blocks (created by our trace generation algorithm), n_1-n_2-...-n_k, where (n_i, n_j) is a flow edge iff $j = i + 1$. The trace represents a frequently executed path in the program (or a copy of such a path in the original CFG to be precise). In addition, n_1, is the only entry to the trace and is called the* **trace header**.

Thus, trace t_1 can only branch into trace t_2 via the header of t_2. As we shall see in Section 4, avoiding side entrances into a trace simplifies energy-oriented optimisations on traces.

A trace can cross the boundaries of functions in both application and library code. In our current implementation, the execution frequency of a path is measured using an edge profile gathered by code instrumentation.

Figure 4 gives our algorithm for constructing the hot traces from the CFG of a binary file. Our example is given in Figure 5. In a block identified by n_f, n is its block number and f its execution frequency. The number alongside an edge (x, y) represents its execution frequency; the number is omitted if the edge is the only outgoing edge of x. In Figure 5, the edge (6,7) introduced by `alto` facilitates the inlining of the call made in block 6. The edges of this kind are ignored in during trace generation. Running our algorithm over Figure 5 produces the CFG shown in Figure 6. There are two traces highlighted in gray boxes, where the trace D6-D8$'$-D9$'$-D11$'$-D7-D1 crosses function boundaries. More details will be explained in Section 3.2.

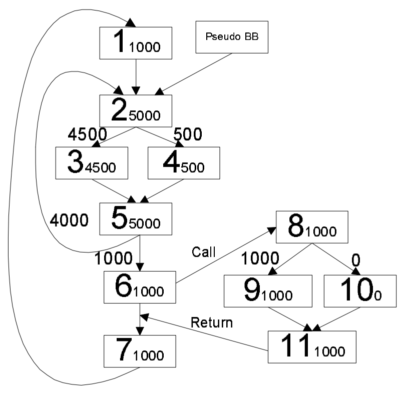

Figure 5: An example CFG

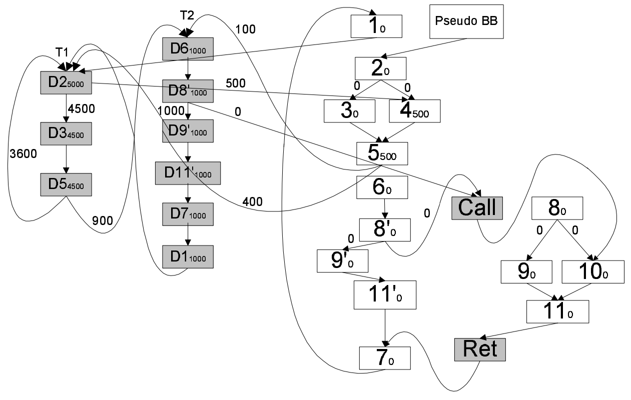

Figure 6: The CFG from Figure 5 with traces.

3.1 Algorithm

Some notations and terminologies are in order. If x is a basic block or an edge, then $x.count$ denotes its execution frequency. If e is an edge, then $e.source$ ($e.sink$) represents the block at the source (sink) of e. At any time, *curtrace* denotes the trace being constructed, with its first block, i.e., trace header denoted by *curtrace.head* and last block by *curtrace.tail*. As a trace grows, a block that joins the trace is always added to the trace as the currently last block. A block in a trace is called a *trace exit* if it has at least two successor blocks (i.e., at least one not in this trace).

Our algorithm, GenTrace, performs three major tasks:

Starting a Trace. In line 2, *headerlist* is initialised with all potential trace headers b such that $Hotness(b) \geqslant$ **HOT_BB_MIN**, which is a tunable parameter currently set to 1000. As is clear from line 5, we always pick the trace header from *headerlist* that is the hottest. According to the function *Hotness*, we increase the hotness value of a block that is a successor of a trace exit by 2% and of a loop header by 1%. These increases are cumulative if a block plays the dual roles. This strategy increases the chances for obtaining well-connected traces and prefers to start a trace from a loop header over a function entry block if both have the same execution frequency.

Once a trace has been created, UpdateHeaderList is called in line 12 to do two things. First, some blocks that are no longer hot are removed from *headerlist* (line 24). This can happen since part of its execution frequency may have been allocated to its duplicate in a hot trace. Second, every block that is a successor of a trace exit of *curtrace* but that does not already appear in a trace is added to *headerlist*.

Growing a Trace. In line 5, *header* is the header of a new trace to be constructed. The basic idea is to find a frequently executed path starting from *header* and duplicate it as a trace in the CFG. Separating this trace from the rest of the CFG requires the control flow edges directed into and out of the traces to be added appropriately. The call to DupHeaderInEdges in line 6 simply makes a copy of *header* and moves its incoming edges to the copy (except those directing out of the pseudo block). In line 7, *hottestedge* represents the first edge to grow the trace. In our current implementation, *hottestedge* is simply chosen as the most frequently executed outgoing edge of *header* with a tie being broken randomly. The remaining part of the task is accomplished mainly in the **while** loop beginning in line 8 in an iterative manner. This entire task will be illustrated using an example in Section 3.2.

If *hottestedge* represents a call edge (e.g., (6, 8) in Figure 5), we will rely on the procedure InlineCriticalPaths available in `alto` [18] to inline a frequently executed subgraph rooted at the entry block of the callee. Afterwards, our algorithm will continue to grow the trace on the inlined subgraph as usual.

Terminating a Trace. In line 8, TraceEnd is called to determine if we should stop growing *curtrace* from its last block *curtrace.tail* along *hottestedge*. One of the four termination conditions applies: (1) the frequency of *curtrace.tail* has dropped below **HOT_BB_PROB** (a tunable parameter set to 50% in our implementation) of the frequency of *curtrace.head*, (2) *hottestedge* represents an unknown indirect jump into the pseudo block, (3) *hottestedge.sink* is the header of an already generated trace, and (4) *hottestedge.sink* is the exit block of the CFG for the program.

Techniques such as value profiling [4] can help resolve some jump targets in our current implementation. However, we have decided not to do so for two reasons. First, the traces that end with some unknown indirect jumps represent only less than 1% of the total number of traces in a program on average. Second, these traces are often less frequently executed than the others.

3.2 Example

Let us illustrate our algorithm using the example given in Figure 5. In line 2, *headerlist* is initialised with the three potential trace headers: blocks 1, 2 and 8. In line 5, $header = 2$ is removed from *headerlist* so that $headerlist = \{1, 8\}$ afterwards. We are ready to build a trace, denoted *curtrace* (line 4), with block 2 as its header. Figure 7(a) depicts the effect of calling DupHeaderInEdges in line 6. Note that the incoming edge from the pseudo block to block 2 is not reallocated to the duplicated block D2 (in gray). In line 7, $hottestedge = (2, 3)$, highlighted in Figure 7(a), is chosen (which is more frequently executed than the other branch).

In line 9, GrowingTrace is called to grow the trace along $hottestedge$=(2,3). The result of this call is illustrated in

```
PROCEDURE GenTrace()
Initialise headerlist with loop headers or function entry
blocks b such that Hotness(b) ⩾ HOT_BB_MIN
while headerlist is not empty
    curtrace = newtrace()
    header = block b with the largest
    Hotness(b) removed from headerlist
    DupHeaderInEdges(curtrace, header)
    hottestedge = HottesttOutEdge(header)
    while !TraceEnd(curtrace, hottestedge)
        GrowingTrace(curtrace, hottestedge)
        hottestedge = HottestOutEdge(hottestedge.sink)
    DupTailOutEdges(curtrace, hottestedge.source)
    UpdateHeaderList(headerlist)

PROCEDURE DupHeaderInEdges(curtrace, header)
dupheader = a copy of header
Remove all incoming edges $e_1, \ldots, e_n$ of header
that are not directing out of the pseudo block, and attach
them to dupheader as its incoming edges (Figure 3)
dupheader.count = $e_1$.count + ⋯ + $e_n$.count
AppendtoTrace(curtrace, dupheader)

PROCEDURE DupTailOutEdges(curtrace, tail)
if the only successor of tail is the pseudo block
    Add an edge from curtrace.tail to the pseudo block
    return
DupOutEdges(curtrace.tail, tail, NULL)

PROCEDURE UpdateHeaderList(curtrace, headerlist)
Remove every block b from headerlist such that
        Hotness(b) < HOT_BB_MIN
for every successor block b of every trace exit in curtrace
    if b is neither a block in curtrace nor a trace header
        Add b to headerlist if Hotness(b) ⩾ HOT_BB_MIN

PROCEDURE GrowingTrace(curtrace, hottestedge)
if hottestedge represents a call edge
    InlineCriticalPaths()
    return
current = curtrace.tail
dupblock = a copy of hottestedge.sink
Add an edge h from current to dupblock
h.count = current.count * $\frac{hottestedge.count}{hottestedge.source.count}$
dupblock.count = h.count
DupOutEdges(current, hottestedge.source, hottestedge)
AppendtoTrace(curtrace, dupblock)

PROCEDURE DupOutEdges(current, original,
    hottestedge) // current is the duplicate of original in curtrace
for every outgoing edge e of original
    hot_fraction$_e$ = current.count * $\frac{e.count}{original.count}$
    if e ≠ hottestedge
        Add an edge e′ from current to e.sink
        e′.count = hot_fraction$_e$
    e.count −= hot_fraction$_e$
original.count −= current.count

FUNCTION TraceEnd(curtrace, hottestedge)
if $\frac{curtrace.tail.count}{curtrace.head.count}$ < HOT_BB_PROB
    return TRUE
elif hottestedge.sink is the pseudo block, a trace header
     or the exit block of the CFG
    return TRUE
else
    return FALSE

FUNCTION Hotness(block)
hot_value = block.count
if block is a successor of a trace exit
    hot_value += 0.02 * HOT_BB_MIN
if IsLoopHeader(block)
    hot_value += 0.01 * HOT_BB_MIN
return hot_value
```

Figure 4: A static trace generation algorithm.

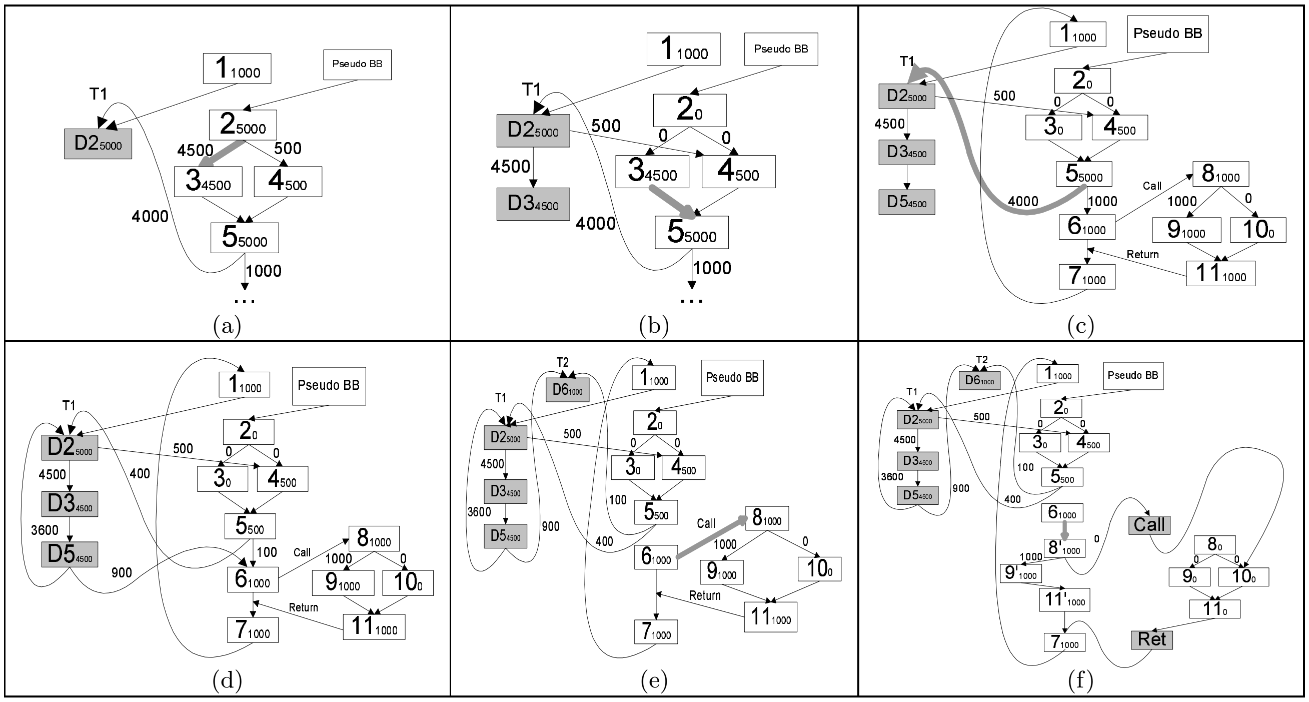

Figure 7: An illustration of GenTrace using the example CFG from Figure 5.

Figure 7(b). In addition, *hottestedge* $= (3, 5)$ chosen in line 10 is also highlighted. In the second iteration of the **while** loop beginning in line 8, the trace grows as shown in Figure 7(c). The direction to grow the trace next is *hottestedge* $= (5, \mathrm{D2})$. But D2 is a trace header. So TraceEnd will return true in the third iteration of the **while** loop beginning in line 8. In line 11, we call DupTailOutEdges to get Figure 7(d), which evolves from Figure 7(c) with the two outgoing edges from block 5 being duplicated for block D5. The first trace D2-D3-D5 has been created.

In line 12, UpdateHeaderList is called with *headerlist* $= \{1, 8\}$. Line 26 has no effect on *headerlist*. In lines 25 – 27, we note that there is one trace already constructed (Figure 7(d)). This trace has two exits, D2 and D5. D2 has one successor 4 outside the trace but $Hotness(4) = 500 + 0.02 * \textbf{HOT\_BB\_MIN} = 520 < \textbf{HOT\_BB\_MIN}$. D5 also has one successor 6 outside the trace. Since $Hotness(6) = 1000 + 0.02 * \textbf{HOT\_BB\_MIN} = 1020 > 1000$, block 6 is included in *headerlist* in line 27, making *headerlist* $= \{1, 8, 6\}$.

We now proceed to build the second trace. In line 5, *header* $= 6$. Calling DupHeaderInEdges in line 6 yields what is shown in Figure 7(e). In line 7, *hottestedge* $= (6, 8)$. In line 9, we call GrowingTrace to grow the trace starting from block 6 along (6,8). The result of this call is given in Figure 7(f). Let us look at this call to GrowingTrace closely. Since $(6, 8)$ is a call edge, InlineCriticalPaths is called in line 30. A frequently executed subgraph rooted at the entry block of the callee is found and inlined. This subgraph happens to be the path 8-9-11, which will be inlined, i.e., duplicated as the path $8'$-$9'$-$11'$ in the caller's context. Note that the "Call" and "Ret" blocks are inserted (by `alto`) to switch between the two different contexts.

Once inlining is completed, the call to GrowingTrace returns. In line 10, *hottestedge* $= (6, 8')$, which is a control flow edge and highlighted in Figure 7(f). Then the similar steps as above are repeated so that we eventually obtain the second trace D6-D8$'$-D9$'$-D11$'$-D7-D1 shown in Figure 6.

In line 12, UpdateHeaderList is called with *headerlist* $= \{1, 8\}$. Both blocks will be removed from the list in line 24 since their frequencies have dropped to 0. Our algorithm terminates since *headerlist* is now empty .

We observe from Figure 6 that blocks 6, $8'$, $9'$ and $11'$ are dead code, which can be removed by dead code elimination.

4. TRACE-BASED OPTIMISATIONS

The goal of this work is to study the effectiveness of a trace-based binary compilation system in supporting energy-oriented optimisations. We have designed and implemented a new optimisation for reducing the functional unit leakage energy for a superscalar architecture, by shutting down unused or infrequently used functional units. We have added this optimisation to the "Traced-based Optimisations" module depicted in Figure 2. We describe this optimisation below.

In current architectures, most of energy consumption is due to switching activity (when hardware components are exercised). However, static power dissipation is projected to be the dominant part of the chip power budget beyond the 0.1 micron feature sizes [5]. The function units, due to its logic circuits used and its high-performance requirement, contribute to a sizeable fraction of the total system leakage power consumption. Therefore, it is beneficial to minimise the static power dissipation by functional units.

```
while (1) {
   n = read(0,abuf,NSAMPLES/2);
   if ( n == 0) break;
   adpcm_decoder(abuf,sbuf,n*2,state);
   write(1,sbuf,n*4);
}
```

Figure 8: Code from `rawcaudio` in Mediabench.

The compiler can identify the program regions in which a functional unit is unused or infrequently used (called the *idle regions*) and communicate this information to the hardware by issuing instructions for turning the unit off at entry points of the idle regions and turning them back on at exits of these regions. The frequent turning on/off activities can consume both significant dynamic energy and execution cycles. To maximise power savings while minimising any adverse impact on performance, it is important to ensure that the idle regions are frequently executed and the insertion points for on/off instructions are infrequently executed.

There are two existing compile-time optimisations for reducing functional unit leakage consumption. Rele *et al.* [19] find the idle regions at the granularity of so-called power blocks for a superscalar architecture while Kim *el al.* [12] choose the idle regions at the granularity of loops for a VLIW architecture. In this paper, we present a traced-based approach for reducing the functional unit leakage consumption. Traces are frequently executed paths, which are inherently inter-procedural and span both user and library functions. In addition, traces are flexible enough to accommodate a variety of flow structures such as recursive calls.

Figure 8 shows a kernel loop extracted from `rawdaudio` in Mediabench. This loop contains the calls to the library functions `read` and `write`. In the absence of any information about the library functions, the two existing techniques [12, 19] assume conservatively that all functional units are required in the library routines. In our link-time compilation framework, the traces can cross these library calls. Therefore, we are capable of deciding if certain functional units are needed or not in the calls to these library routines.

Section 4.1 describes some minimal architectural support required. Section 4.2 presents our trace-based solution, which is illustrated by an example in Section 4.3.

4.1 Architecture support

Techniques such as input vector control (IVC) and supply gating (SG) can be used for runtime leakage control of functional units [5]. Based on such architectural support, we can turn off an unused functional unit to reduce static power dissipation and turn it back on before it will be used.

We assume the availability of on and off instructions in the instruction set architecture (ISA) of the underlying hardware. An on/off instruction always indicates the type of the functional unit that is to be turned on/off. The execution of an on (off) instruction causes the hardware to select a functional unit of the specified type to be turned on (off). The latencies and dynamic energy overheads of on/off instructions depend on the functional unit type and the exact implementation mechanism.

4.2 Trace-based Leakage Optimisation

Once the hot traces for a program have been created, the

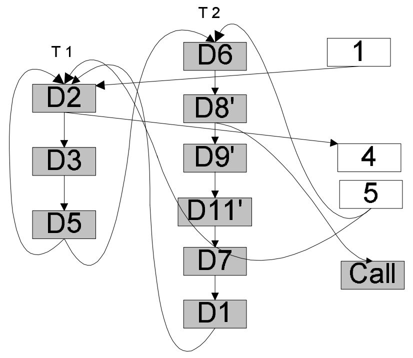

Figure 9: The trace flow graph of Figure 6.

idle regions and their associated insertion points for placing on and off instructions can be easily identified. Informally, an idle region is made up of traces that are connected *directly* by flow edges. Its associated on and off instructions are inserted only at its boundaries. The granularity of an idle region is tunable, ranging from a graph with a single trace to a graph consisting of all possibly connected traces.

Like [12, 19], a functional unit stays either on or off exclusively inside an idle region. In addition, a functional unit is turned off in an idle region only if the amount of instruction level parallelism (ILP) within the region is not adversely affected. This means that all functional units of a given type will be left on in an idle region even if there is sufficient ILP to keep all busy only at a small part of the region. As a consequence, varying the granularity of idle regions allows us to make tradeoffs between energy savings and performance. In general, using smaller idle regions allows more functional units to be turned off even for a shorter period of time, which results in better leakage energy savings in absolute terms. However, increasingly more on and off instructions used will consume more dynamic energy and execution cycles. Therefore, an effective optimisation should ensure that the leakage energy saved exceeds the dynamic energy consumed.

We give a leakage energy optimisation algorithm for a set of f_{num} identical functional units of type f_{type}. Its development relies on a so-called trace flow graph, which is defined below and illustrated in Figure 9 using our running example.

Definition 2. *An edge (x, y) is called (1) a* **trace entry edge** *if x is not in a trace and y is a trace header, (2) a* **trace exit edge** *if x is in a trace but y is not, and (3) a* **trace link edge** *if both x and y are in traces (which may be identical) and y is a trace header.*

Definition 3. *A* **trace flow graph** *is the graph consisting of (1) all the hot traces (including the blocks in these traces and the edges connecting these blocks), and (2) all trace entry, exit and link edges and their incident blocks.*

The trace flow graph of Figure 6 is shown in Figure 9, where $(1, \mathrm{D2})$, $(5, \mathrm{D2})$ and $(5, \mathrm{D6})$ are trace entry edges, $(\mathrm{D2}, 4)$ and $(\mathrm{D8}, \mathrm{Call})$ are trace exit edges, and $(\mathrm{D1}, \mathrm{D2})$, $(\mathrm{D5}, \mathrm{D2})$ and $(\mathrm{D5}, \mathrm{D6})$ are trace link edges. Note that $(\mathrm{D5}, \mathrm{D2})$ is a trace link edge for the same trace, i.e., T_1.

Figure 10 gives our algorithm for reducing leakage energy by f_{num} functional units of type f_{type}. Once the trace flow graph is built (line 2), we call GetMaxILPinTrace to analyse each trace t individually to obtain the maximum number of instructions or operations, $t.ILP$, in the trace t that can execute in parallel on this functional unit type (lines 3 – 4). We can obtain a more accurate estimate of $t.ILP$ if we extract the resource usage information from the instruction scheduling pass in the compiler. In our current implementation targeting out-of-order superscalar architectures, we estimate $t.ILP$ as in lines 9 – 15, where **WIN_SIZE** is the size of the instruction scheduling window. The dependences among instructions falling into a window size are examined to estimate $t.ILP$.

In line 5, all idle regions are found by calling FindIdleRegions. Essentially, an idle region consists of multiple traces that are connected by trace link edges. However, some trace link edges may be infrequently executed. Such edges are ignored (lines 20 – 22) so that we can tune the granularity of idle regions formed. **AFFINITY** is a tunable parameter in the range $[0, 1]$. If it is set to 0, then all idle regions are singleton traces. Such a setting is the most aggressive in turning off unused or infrequently used functional units.

```
 1 PROCEDURE LeakageOpt($f_{type}$, $f_{num}$)
 2 Build the trace flow graph $TFG$ (Definition 3)
 3 for every trace $t$ in $TFG$
 4     $t.ILP$ = GetMaxILPinTrace($t$, $f_{type}$, $f_{num}$)
 5 $SetofRegions$ = FindIdleRegions($TFG$)
 6 for every $r$ in $SetofRegions$
 7     $r.ILP$ = GetMaxILPinRegion($r$)
 8 InsertOnOffInsts
 9 FUNCTION GetMaxILPinTrace($t$, $f_{type}$, $f_{num}$)
10 Let the sequence of instructions in trace $t$ be $I_1, \ldots, I_m$
11 $ILP = 0$
12 for $i = 1$ to $m$
13     $num$ = maximum number of instructions in
              $\{I_i, \ldots, I_{\min(i+\mathbf{WIN\_SIZE}-1,m)}\}$ that can
              execute on $f_{type}$ in parallel
14     $ILP = \max(ILP, num)$
15 return $\min(ILP, f_{num})$
16 FUNCTION GetMaxILPinRegion($r$)
17 Let $t_1 \ldots, t_p$ be all $p$ traces in $r$
18 return $\max(t_1.ILP, \ldots, t_p.ILP)$
19 PROCEDURE FindIdleRegions($TFG$)
20 Let $L$ be the set of all trace link edges $e$
   such that $e.count < (\frac{1}{\mathbf{AFFINITY}} - 1) * \mathbf{HOT\_BB\_MIN}$
21 $TFG'$ = $TFG$ with all edges in $L$ removed
22 return set of all connected subgraphs in $TFG'$
23 PROCEDURE InsertOnOffInsts
24 Insert $f_{num}$ "on instructions" for $f_{type}$ at entry to main
25 for every trace entry edge $(x, y)$ in the program
26     Insert $f_{num} - y.region.ILP$ off instructions on $(x, y)$
27 for every trace exit edge $(x, y)$ in the program
28     Insert $f_{num} - x.region.ILP$ on instructions on $(x, y)$
27 for every trace link edge $(x, y)$ in the program
28     $n = x.region.ILP - y.region.ILP$
28     if $n < 0$
39         Insert $|n|$ on instructions on $(x, y)$
30     elif $n > 0$
31         Insert $n$ off instructions on $(x, y)$
```

Figure 10: A leakage optimisation algorithm.

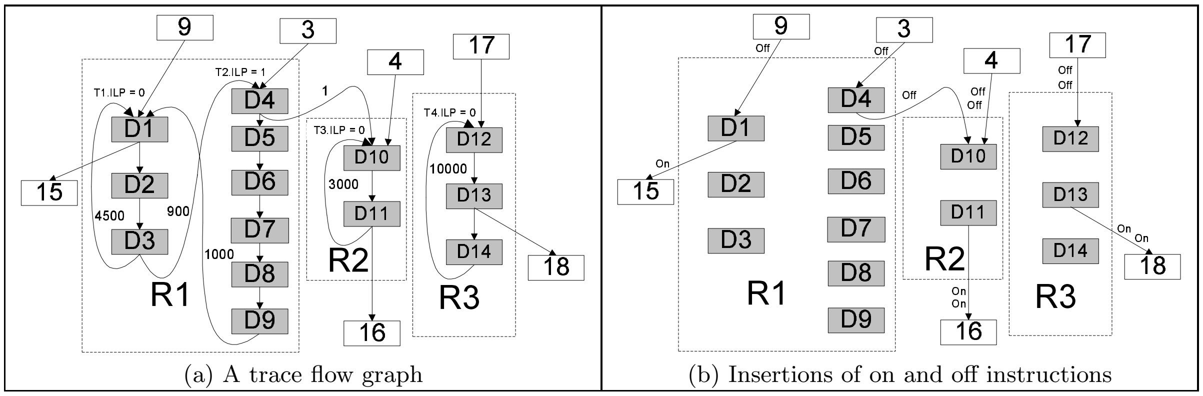

Figure 11: An illustration of LeakageOpt.

If it is set to 1, then every idle region is the largest possible with the largest number of directly connected traces. Such a setting aims at reducing the execution cycles and dynamic energy consumed by the inserted on and off instructions. Varying the value of **AFFINITY** allows tradeoffs to be made between energy savings and performance.

Once idle regions are formed, the resource requirement in each region r is computed based on those of its constituent traces (lines 6 – 7 and 16 – 18). In fact, the number of functional units required by r, denoted $r.ILP$, is simply taken to be the maximum of ILP's among all its traces.

Finally, in line 8, we call InsertOnOffInsts to introduce all required on and off instructions at trace entry, exit and link edges, which are infrequently executed since they are not inside traces. In line 24, we assume that all functional units are on outside traces, since the power savings should really come from the idle regions. In lines 25 – 26, we turn off unused functional units by inserting the specified number of off instructions on the trace entry edges and turn them back on at the trace exit edges (lines 27 – 28). The treatment for trace link edges are self-explanatory if we note that such an edge always connects two idle regions (which can be identical). In this procedure, $x.region$ and $y.region$ denote the idle regions in which the two incident blocks belong to, respectively (when they are applicable).

4.3 Example

Let us continue to assume that **HOT_BB_MIN**=1000 and **HOT_BB_PROB**=50%. In LeakageOpt, let us assume that **AFFINITY** = 0.5. The value of the other parameter **WIN_SIZE** is irrelevant here. For the running example given in Figure 6, there will be just one idle region consisting of both traces. So we have decided to illustrate LeakageOpt using a different example given in Figure 11.

Suppose that the trace flow graph obtained in line 2 is shown in Figure 11(a). Suppose that there are $f_{num} = 2$ functional units of a particular type under consideration. There are four traces, $T_1, \ldots, T_4$ with their ILP values as shown. The frequencies of all six trace link edges are given. In line 5, FindIdleRegions is called. $L = \{(\mathrm{D4}, \mathrm{D10}\}$ in line 20 since $(\mathrm{D4}, \mathrm{D10})$ is infrequently executed. By removing this edge from the trace flow graph (lines 21 – 22), we will be left with three connected subgraphs. As a result, the three idle regions, $R_1, \ldots, R_3$ are found, which are indicated by the dotted boxes in Figure 11, where $R_1.ILP = \max(T_1.ILP, T_2.ILP) = 1$, $R_2.ILP = T_3.ILP = 0$ and $R_3.ILP = T_4.ILP = 0$. The on and off instructions are inserted as shown in Figure 11(b), where the irrelevant edges in the trace flow graph are suppressed for clarity.

Configuration	**HOT_BB_MIN**	**HOT_BB_PROB**
DEFAULT	1000	50%
CONFIG1	100	50%
CONFIG2	1000	25%
CONFIG3	5000	50%
CONFIG4	10000	25%

Table 1: Five settings for HOT_BB_MIN and HOT_BB_PROB used in evaluating GenTrace.

5. EXPERIMENTAL RESULTS

In our experiments, we evaluate the effectiveness of our trace generation algorithm in identifying the hot traces and the effectiveness of our leakage optimisation algorithm in reducing the functional unit leakage energy.

We use seven benchmarks from Mediabench [17]. All the benchmarks are compiled using `gcc` at the optimisation level "O2". The profiling information is obtained using the so-called "second data set" in [17]. All the benchmarks are executed using the data set that comes with the benchmarks.

We consider a superscalar out-of-order architecture consisting of 2 integer multipliers, 4 integer ALUs for non-multiplication integer operations, 1 floating point multiplier and 4 floating point adders. We use simoutorder, an out-of-order cycle-level simulator from SimpleScalar [21].

5.1 Trace Generation: GenTrace

The two metrics are used: (1) the percentage of the execution time (in cycles) spent on executing the instructions in the traces and (2) the percentage (static) instruction count increase due to the duplication of the traces.

GenTrace has two tunable parameters: **HOT_BB_MIN** and **HOT_BB_PROB**. The traces it generates can vary depending on the values used for the two parameters. We evaluate GenTrace below using the five configurations listed in Table 1, where DEFAULT is the default setting.

Figure 12 demonstrates the quality of the traces generated. These results indicate that GenTrace is capable of capturing the majority of the hot traces in these benchmarks. In Figure 13, we observe that duplicating the hot traces in a program has resulted in only a small increase in the total number of instructions generated (from both application and library code) for all the benchmarks except `toast`. This happens because the most computations in an embedded application tend to concentrate on small hot portions of the application.

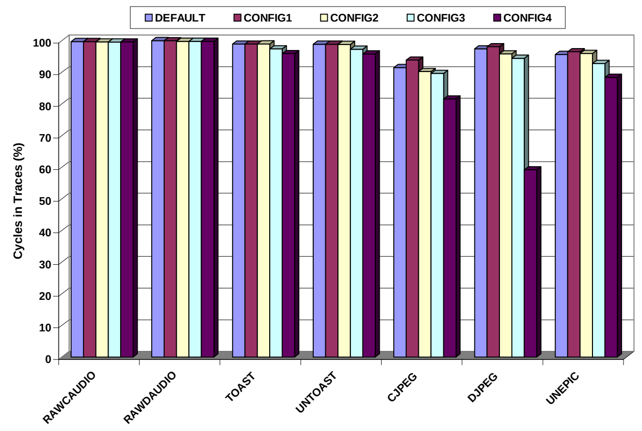

Figure 12: Percentage of cycles spent on hot traces.

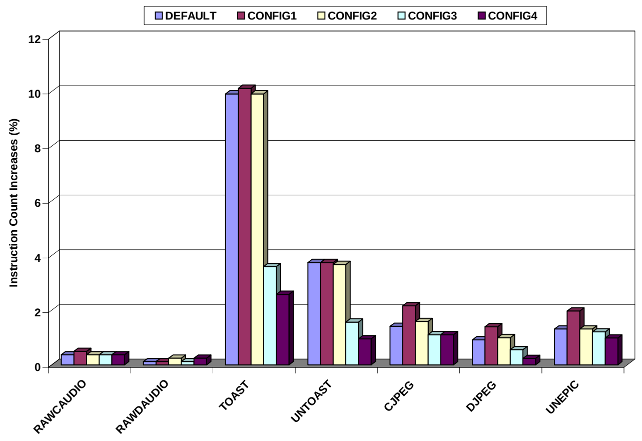

Figure 13: Instruction count increases.

Overall, our default configuration for the two parameters **HOT_BB_MIN** and **HOT_BB_PROB** is a reasonable choice in terms of the two metrics used, at least for the benchmarks used in our experiments.

5.2 Leakage Optimisation: LeakageOpt

We evaluate LeakageOpt by applying it to the 2 integer multipliers only since it is equally applicable to any other functional unit type in the architecture. We assume that supply (or power) gating (SG) is used to turn off a functional unit. According to [3], the leakage energy saving for SG is 100% and several cycles are sufficient to turn on/off a functional unit. In our experiments, the latency for an on or off instruction is assumed to be 10 cycles.

Following [12], the percentage leakage energy reduction on the 2 integer multipliers is quantified by $\frac{M_1+M_2}{2*C}*$**LS_RATIO**, where M_k is the total number of cycles that the k-th unit is off, C is the total number of execution cycles in the program and **LS_RATIO** is the percentage leakage energy saving per cycle when a functional unit is off. **LS_RATIO** is set to 100% since SG is used. M_k is computed conservatively such that it does not include the on-to-off or off-to-on transition cycles elapsed in a functional unit. Finally, the architecture we consider supports control flow speculation through branch prediction. Some instructions that are enclosed between a pair of off and on instructions may be speculatively executed before the off instruction. If the speculation turned out to be correct, the cycles spent on speculatively executing these instructions are not included in M_k (since they are no longer in the idle region in the dynamic sense).

In evaluating LeakageOpt, we use the default configuration for GenTrace, i.e., we set **HOT_BB_MIN** = 1000 and **HOT_BB_PROB** = 50%. LeakageOpt has two parameters: **WIN_SIZE** and **AFFINITY**. For the architecture we consider, **WIN_SIZE** = 16. The tunable parameter **AFFINITY** takes three different values: 0, 1/1.5 and 1.

Figure 14 shows that we can achieve significant leakage energy savings for the benchmarks used. The leakage energy improvement for a benchmark with a particular **AFFINITY** value is given by $\frac{M_1+M_2}{2*C} * 100\%$, where $M_1 + M_2$ is the total number of cycles that the two multipliers are off and C is the total number of execution cycles for the benchmark (added with traces). The three bars for a benchmark displayed from left to right are associated with 0, 1/1.5 and 1, respectively. As **AFFINITY** increases, some idle regions that are not connected previously may become connected. This implies that the on and off instructions required can only be reduced. As a result, when **AFFINITY** increases, both C and $M_1 + M_2$ in $\frac{M_1+M_2}{2*C} * 100\%$ tend to decrease, which may cause $\frac{M_1+M_2}{2*C} * 100\%$ to fluctuate. In Figure 14, there are small variations associated with the three bars for almost every benchmark. The main reason is that a number of idle regions in these benchmarks do not have multiplications. If an idle region contains no multiplications, breaking it into smaller idle regions will not cause any extra on and off instructions to be introduced by LeakageOpt.

Figure 15 shows the performance degradations due to the introduction of the on/off instructions. Table 2 gives the average duration for which a multiplier is off. The performance degradations are small for all the benchmarks. As is clear from Figure 15, each benchmark runs increasingly no slower as **AFFINITY** increases from 0 to 1/1.5 to 1. In the case of `toast`, the off duration when **AFFINITY** = 0 is less than half of the off duration when **AFFINITY** = 1/1.5 or 1. The performance degradation is the worst when **AFFINITY** = 0. A similar pattern can be observed in `cjpeg` and `djpeg`. The benchmark `untoast` enjoys 57.97% leakage energy reduction under the three different **AFFINITY** values. But it runs 1.31% slower when **AFFINITY** = 0 and only 0.97% slower when **AFFINITY** = 1/1.5 or 1. This implies that while the two multipliers are shut down more frequently when **AFFINITY** = 0, the total execution time of the program becomes longer due to the more on and off instructions used. Overall, the relative leakage energy savings in all the three cases are about the same. Finally, each of the remaining three benchmarks, `rawcaudio`, `rawdaudio` and `unepic`, suffers the same performance penalty under the three different **AFFINITY** values.

As shown in Table 2, the off durations are quite long for most benchmarks. It is projected that in future microprocessors static power consumption will dominate the total power consumption [5]. The large off durations will reduce the dynamic energy consumed in the turning on/off activities, leading to overall energy savings for these benchmarks.

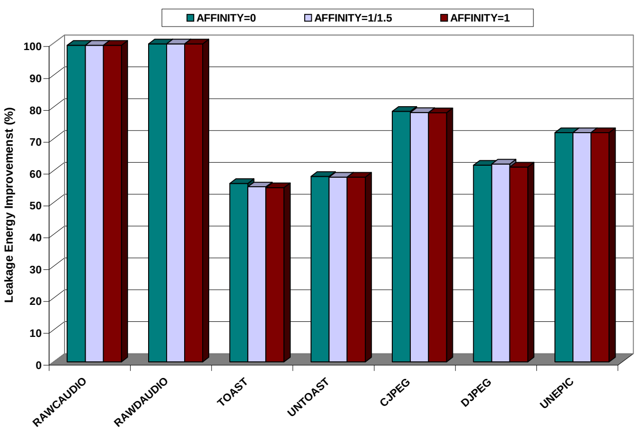

Figure 14: Percentage leakage energy reductions relative to the programs added with traces.

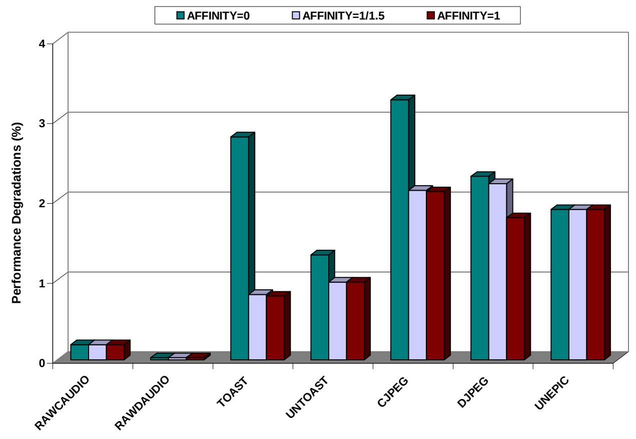

Figure 15: Percentage increases in execution cycles relative to the programs added with traces.

Our experimental results suggest that **AFFINITY** = 1 appears to be a good choice for the purposes of reducing leakage energy consumption on the two multipliers for these benchmarks. In comparison with the other two **AFFINITY** values, the leakage energy savings obtained are about the same but the performance slowdowns are the smallest.

6. RELATED WORK

There has been some recent work on exploring suitable compiler infrastructures for embedded systems. Kulkarni *et al.* [13] emphasise the importance of an interactive environment that allows the user to tune the sequence of optimisation phases to meet her/his design goals. They demonstrate how iteratively applying optimisation phases can have a significant impact on static and dynamic instruction counts for the SPARC architecture. Zhao *et al.* [27] describe a framework for tackling the phrase-ordering problem based on predicting the impact of an optimisations on performance and resource usage. They present an instance of this framework and demonstrate the performance benefits of selectively applying loop transformations based on estimating the impact of an optimisation on cache misses. Kadayif *et al.* [10] present a framework for estimating analytically the energy consumption of a high-level code segment in order to guide high-level code optimisations. While these frameworks focus on high-level optimisations, ours permits complementary optimisations to be carried out on binaries at link time.

There are a number of binary translation systems around [1, 6, 22]. These systems aim at improving performance or otherwise achieving portability while ours is designed with both performance and energy consumption in mind.

Traces are not new. Trace scheduling [7] is a well-known technique for increasing the amount of ILP by scheduling a sequence of basic blocks together, which typically represents a frequently executed path in the program. Traces have a number of extensions such as hyperblocks [16] and regions [9]. In Dynamo [1], the frequently executed paths are identified at run time so as to improve the program performance transparently. These previous works show that a trace-based approach is effective in supporting performance-oriented optimisations. To the best of our knowledge, this work is the first to demonstrate that the hot traces represent a suitable framework to support energy-oriented optimisations as well.

Our trace generation algorithm identifies the hot traces across procedural boundaries at link time based on an interprocedural CFG constructed from a binary file. This CFG is imprecise since the targets of some jumps may be unknown or even *illegal* since a branching instruction in one function may jump to the middle of another function. These problems do not exist when the traces are constructed at compile time [7, 9, 16] or cause less trouble when the traces are constructed at run time [1].

Reducing energy consumption is important for embedded devices. Compiler optimisations can play an important role due to the need to meet conflicting constraints on time, code size and energy consumption. In the absence of architectural support, compiler techniques can improve the dynamic energy behaviour of a program in many phases of the compilation process, such as instruction selection [15], register allocation [8] and instruction scheduling [14]. Many high-level code transformations such as loop tiling reduce the cache misses in the program, and consequently, the dynamic energy spent on cache [11].

By exploiting available architectural support in an embedded system, the compiler can generate code to dynamically reconfigure the processor resources to make tradeoffs between performance and energy usage. For example, Saputra *el al.* [20] explore DVS as a means of improving the dynamic energy consumption of a program without increasing its execution time. Xie *et al.* [25] analyse and evaluate the opportunities and limits of compile-time DVS scheduling. To reduce static power dissipation, the compiler can generate instructions to turn off unused or infrequently used

Benchmark	**AFFINITY** = 0	**AFFINITY** = 1/1.5	**AFFINITY** = 1
RAWCAUDIO	3064	3064	3064
RAWDAUDIO	22070	22070	22070
TOAST	523	1115	1166
UNTOAST	553	549	549
CJPEG	1007	1105	1107
DJPEG	2468	2773	3076
UNEPIC	1169	1169	1169

Table 2: Average "off duration" (in cycles).

components. Zhang *et al.* [26] reduce data cache leakage energy by turning off cache lines when they are not used and turning them on just before they are accessed later. Rele *et al.* [19] reduce the leakage energy spent on functional units by turning off unused or infrequently used functional units for a superscalar architecture. The greedy nature of this approach may introduce many off-on instruction pairs that are too close together. These spurious off-on pairs are expected to be nullified dynamically by the hardware to avoid the dynamic energy that might otherwise be consumed during the off-to-on transitions. Kim *et al.* [12] study the same optimisation problem for a VLIW architecture. In this work, we provide a trace-based approach for reducing static power dissipation by functional units at link time.

7. CONCLUSION

In this paper, we present a trace-based binary compilation framework for energy-aware computing. We have implemented one such a framework in `alto`, a link-time optimiser for the Alpha architecture. Two components we have added to `alto` are a trace generator and a trace-based optimiser. Our trace generation algorithm works by identifying frequently executed paths in a CFG and duplicating them as the (hot) traces in the CFG. Separating traces from non-traces has one important benefit. Traces have simple structures: every trace has only a single entry point. This makes it easy to implement many optimisations in our trace-based framework. Our algorithm in identifying traces is both effective and practical. Our experimental results over benchmarks indicate that the most execution cycles are spent on the traces, which cause only small code size increases.

In our binary compilation framework, the traces are inherently inter-procedural and span functions in both application and library code. In addition, the traces naturally accommodate various kinds of control flow structures such as recursive calls. To evaluate the effectiveness of using traces to support energy-oriented optimisations, we have developed a new leakage optimisation for functional units on traces. Our algorithm is simple since traces allow the idle regions and on/off insertion points to be identified easily. It is also effective since significant leakage energy reductions can be obtained for benchmarks at small performance penalties.

Our trace-based algorithm can be generalised to deal with a number of compiler-directed optimisation problems in embedded applications. Two more examples are the problem for reducing cache leakage energy [26] and the problem for employing cache locking to improve WCET estimates [23]. These problems share the same solution pattern as the problem of reducing functional unit leakage: the compiler identifies the regions in which an optimisation is done and inserts power-aware instructions at some suitable points to instruct the architecture to turn on/off the specified hardware feature. All these tasks can be done similarly in our framework.

8. REFERENCES

[1] V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: a transparent dynamic optimization system. In *ACM SIGPLAN '00 Conference on Programming Language Design and Implementation*, pages 1 – 12, Vancouver, British Columbia, Canada, 2000. ACM Press.

[2] S. Borkar. Design challenges of technology scaling. *IEEE Micro*, 19(4):23–29, July 1999.

[3] J. A. Butts and G. S. Sohi. A static power model for architects. In *33rd Annual ACM/IEEE International Symposium on Microarchitecture*, pages 191–201. ACM Press, 2000.

[4] B. Calder, P. Feller, and A. Eustace. Value profiling. In *International Symposium on Microarchitecture*, pages 259–269, 1997.

[5] A. Chandrakasan, W. J. Bowhill, and F. Fox. *Design of High-Performance Microprocessor Circuits*. IEEE Press, 2001.

[6] J. C. Dehnert, B. K. Grant, J. P. Banning, R. Johnson, T. Kistler, A. Klaiber, and J. Mattson. The transmeta code morphing software: using speculation, recovery, and adaptive retranslation to address real-life challenges. In *1st ACM/IEEE International Symposium on Code Generation and Optimization*, pages 15–24. IEEE Computer Society, 2003.

[7] J. Fisher. Trace scheduling: a technique for global microcode compaction. In *IEEE Transactions on Computers*, pages 478–490, 1981.

[8] C. H. Gebotys. Low energy memory and register allocation using network flow. In *34th Annual Conference on Design Automation Conference*, pages 435–440. ACM Press, 1997.

[9] R. E. Hank, W.-M. Hwu, and B. R. Rau. Region-based compilation: an introduction and motivation. In *28th ACM/IEEE International Symposium on Microarchitecture*, pages 158–168. IEEE Computer Society Press, 1995.

[10] I. Kadayif, M. Kandemir, N. Vijaykrishnan, M. J. Irwin, and A. Sivasubramaniam. Eac: a compiler framework for high-level energy estimation and optimization. In *5th Design Automation and Test in Europe Conference*, pages 4–8, 2002.

[11] M. T. Kandemir, N. Vijaykrishnan, M. J. Irwin, and W. Ye. Influence of compiler optimizations on system power. In *Design Automation Conference*, pages 304–307, 2000.

[12] H. S. Kim, N. Vijaykrishnan, M. Kandemir, and M. J. Irwin. Adapting instruction level parallelism for optimizing leakage in VLIW architectures. In *ACM SIGPLAN '03 Conference on Languages, Compilers, and Tools for Embedded Systems*, pages 275 – 283, San Diego, California, USA, 2003. ACM Press.

[13] P. Kulkarni, W. Zhao, H. Moon, K. Cho, D. Whalley, J. Davidson, M. Bailey, Y. Paek, and K. Gallivan. Finding effective optimization phase sequences. In *ACM SIGPLAN '03 Conference on Languages, Compilers, and Tools for Embedded Systems*, pages 12–23. ACM Press, 2003.

[14] C. Lee, J. K. Lee, T. Hwang, and S.-C. Tsai. Compiler optimization on instruction scheduling for low power. In *13th International Symposium on System Synthesis*, pages 55–60, Madrid, Spain, 2000. ACM Press.

[15] M. Lorenz, L. Wehmeyer, and T. Dräger. Energy aware compilation for DSPs with SIMD instructions. In *ACM SIGPLAN' 02 Conference on Languages, Compilers, and Tools for Embedded Systems*, pages 94–101. ACM Press, 2002.

[16] S. A. Mahlke, D. C. Lin, W. Y. Chen, R. E. Hank, and R. A. Bringmann. Effective compiler support for predicated execution using the hyperblock. In *25th ACM/IEEE International Symposium on Microarchitecture*, pages 45–54. IEEE Computer Society Press, 1992.

[17] Mediabench. http://cares.icsl.ucla.edu/MediaBench/applications.html.

[18] R. Muth. *ALTO: A Platform for Object Code Modification*. PhD thesis, The University of Arizona, 1999.

[19] S. Rele, S. Pande, S. Onder, and R. Gupta. Optimizing static power dissipation by functional units in superscalar processors. In *11th International Conference on Compiler Construction*, pages 261 – 275. Springer-Verlag, 2002.

[20] H. Saputra, M. Kandemir, N. Vijaykrishnan, M. J. Irwin, J. S. Hu, C.-H. Hsu, and U. Kremer. Energy-conscious compilation based on voltage scaling. In *ACM SIGPLAN '02 Conference on Languages, Compilers, and Tools for Embedded Systems*, pages 2 – 11, Berlin, Germany, 2002. ACM Press.

[21] Simplescalar. http://www.simplescalar.com.

[22] D. Ung and C. Cifuentes. Machine-adaptable dynamic binary translation. In *ACM SIGPLAN Workshop on Dynamic and Adaptive Compilation and Optimization*, pages 41–51. ACM Press, 2000.

[23] X. Vera, B. Lisper, and J. Xue. Data caches in multitasking hard real-time systems. In *24th IEEE International Real-Time Systems Symposium*, pages 154 –165, 2003.

[24] L. Wehmeyer, M. Jain, S. Steinke, P. Marwedel, and M. Balakrishnan. Analysis of the influence of register file size on energy consumption, code size and execution time. *IEEE Transactions on Computer-Aided Design*, 20(11), Novemver 2001.

[25] F. Xie, M. Martonosi, and S. Malik. Compile-time dynamic voltage scaling settings: opportunities and limits. In *ACM SIGPLAN' 03 Conference on Programming Language Design and Implementation*, pages 49–62. ACM Press, 2003.

[26] W. Zhang, M. Karakoy, M. Kandemir, and G. Chen. A compiler approach for reducing data cache energy. In *17th Annual International Conference on Supercomputing*, pages 76 – 85, San Francisco, CA, USA, 2003. ACM Press.

[27] M. Zhao, B. Childers, and M. L. Soffa. Predicting the impact of optimizations for embedded systems. In *ACM SIGPLAN '03 Conference on Languages, Compilers, and Tools for Embedded Systems*, pages 1–11. ACM Press, 2003.