# Trace-Based Data Cache Leakage Reduction at Link Time

Lian Li[1,2] and Jingling Xue[1,2]

[1] Programming Languages and Compilers Group, School of Computer Science and Engineering, University of New South Wales, Sydney, NSW 2052, Australia
[2] National ICT Australia

**Abstract.** This paper investigates the benefits of conducting leakage energy optimisations for data caches at link time for embedded applications. We introduce an improved algorithm for identifying and constructing the traces in a binary program and present a trace-based optimisation for reducing leakage energy in data caches. Our experimental results using Mediabench benchmarks show that good leakage energy savings can be achieved at the cost of some small performance and code size penalties. Furthermore, by varying the granularity of optimisation regions, which is a tunable parameter, embedded application programmers can make the tradeoffs between energy savings and these associated costs.

## 1 Introduction

Leakage power dissipation is estimated to be around 10-15% of the total power dissipation in high-speed processes [6] and this fraction is projected to be the dominant part of the chip power budget beyond the 0.1 micron feature sizes [2]. Leakage energy consumption in caches is particularly significant since they contain a significant fraction of the on-chip transistors in a microprocessor. It is projected that leakage will represent more than 70% of the energy consumed in caches if left unchecked for the 0.07 micron process [10]. Therefore, reducing leakage energy for caches is of practical importance in modern microprocessors.

In our earlier work [12], we introduced a trace-based, link-time compilation framework for embedded systems and reported its benefits in reducing leakage energy on functional units. In this work, we investigate the benefits of supporting leakage energy optimisations on data caches in such a framework. In particular, we present an improved algorithm for constructing the traces in a binary program. Based the traces thus generated, we introduce a trace-based optimisation for reducing leakage energy on data caches. We present experimental evaluations of our optimisation using Mediabench benchmarks.

Guided by some execution profiling information, the frequently executed paths in a binary program are identified and duplicated as single-entry traces. Separating frequently from infrequently executed paths (spanning both user and library functions) at link time enables the compiler to focus energy optimisations on the hot traces (i.e., spots) across the whole program. The traces are further connected to form the so-called optimisation regions, where their entries and exits are less frequently executed than what are inside. To reduce the leakage energy on a cache in an optimisation region, the

C. Jesshope and C. Egan (Eds.): ACSAC 2006, LNCS 4186, pp. 175–188, 2006.

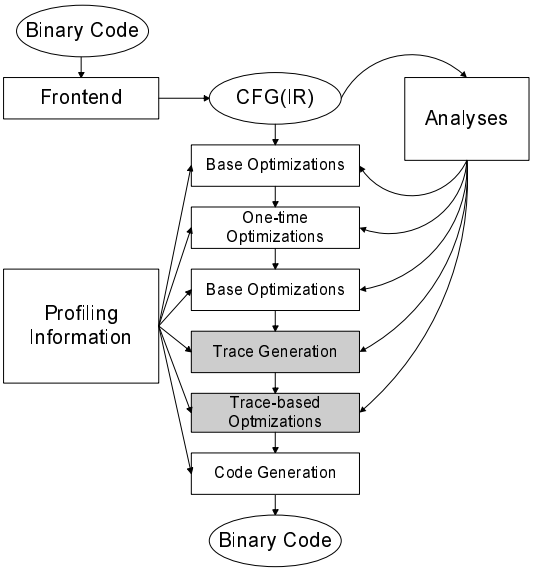



**Fig. 1.** A traced-based, link-time framework implemented in `alto` for the Alpha architecture

compiler invokes an appropriate architectural feature at the entries of the region to put the cache in an energy-saving mode and then restores the cache to its normal mode at its exits. Our experimental results using Mediabench programs show that significant leakage energy savings can be obtained at the cost of small execution time and code size increases. In addition, varying the granularity of optimisation regions makes it possible to make the tradeoffs between energy savings and these associated costs.

The rest of this paper is organised as follows. Section 2 introduces our trace-based methodology. In particular, we discuss an improved algorithm for identifying and constructing the traces in a binary program. Section 3 presents a traced-based optimisation for reducing leakage energy on caches. In Section 4, we evaluate this work with Mediabench benchmarks. Section 5 reviews the related work. Section 6 concludes the paper.

## 2 Trace-Based Methodology

Figure 1 depicts our trace-based framework for supporting energy-oriented optimisations on binaries. We have implemented it `alto`, a link-time optimiser for the Alpha architecture [15]. The two components we have added to `alto` are highlighted by the two boxes in gray. We refer to [12] for a description of the functionalities of all the components in the framework. Our framework supports static binary optimisations. The advantage is that no runtime system is needed. However, applications that use shared or runtime libraries cannot be handled. In addition, static binary translators such as `alto` [15] rely on the reallocation information from the linker to reconstruct a CFG from a binary file. So all relocatable addresses in the file must be identifiable.

```
1  #DEFINE BB_THRESHOLD = 5%
2  #DEFINE BB_MIN = the execution frequency of basic block b_i such that i
       is the largest satisfying: BB_THRESHOLD ⩾ (i/N) × 100(%), where
       b_1, ..., b_N are the N basic blocks in the program sorted in the non-increasing
       order of their execution frequencies
3  #DEFINE BB_PROB = 50%

4  Boolean FUNCTION Hot(block)
5  return block.freq ⩾ BB_MIN × BB_PROB

6  PROCEDURE GenTrace()
7  Initialise headerlist with loop headers or
   function entry blocks h such that
           h.freq ⩾ BB_MIN
8  while headerlist is not empty
9      header = block h removed from headerlist such that h.freq is the largest (by
           favouring a tieing candidate that is a successor of a trace exit in order
           to create well-connected traces)
10     Identify the trace starting from header h
11     Duplicate the trace in the program
12     UpdateHeaderList(headerlist)

13 PROCEDURE UpdateHeaderList(headerlist)
14 Remove every block b from headerlist
       such that Hot(b) does not hold
15 for every successor block s of a trace exit
16     if s is not in a trace such that Hot(s) holds
17         Add s to headerlist
```

**Fig. 2.** A static trace generation algorithm

Let us present an improved algorithm of [12] for identifying and constructing the traces in a binary program. A *trace* is a frequently executed path in a binary program. Such a trace may cross function boundaries. The first (basic) block in a trace is called a *trace header*. A block in a trace is called a *trace exit* if it has one successor block that is not in a trace. Based on profiling information, the frequently executed paths in the CFG of a program are identified and duplicated as single-entry traces. Thus, a trace $t_1$ can only branch into a trace $t_2$, where $t_1$ and $t_2$ may be identical, via the trace header of $t_2$. Single-entry traces allow compiler optimisations to be easily applied. In [12], we presented an algorithm for constructing the traces in binaries. We give a high-level sketch of that algorithm in Figure 2 and describe three improvements we have made.

Our algorithm identifies and builds the hot traces in a program by making use of three profiling-related parameters, which are defined in lines 1 – 3. In fact, BB_THRESHOLD is introduced only to define BB_MIN, which, together with BB_PROB, are used explicitly in our algorithm. These three parameters serve the following purposes. Initially, our algorithm starts with loop headers or function entries $b$ that are potential trace headers only when $b.freq \geqslant$ BB_MIN, where $b.freq$ is the (profiled) execution frequency of block $b$ (line 7). When a trace grows, the blocks that join the trace become progressively

non-larger in terms of their execution frequencies. However, every block that appears in a trace must be hot. A block $b$ is *hot* if the predicate *Hot*($b$) defined in lines 4 – 5 evaluates to true, i.e., if $b.freq \geqslant$ BB_MIN $\times$ BB_PROB. In addition, a block $b$ does not belong to a trace if its execution frequency has dropped below BB_PROB(%) of the execution frequency of the header of that trace.

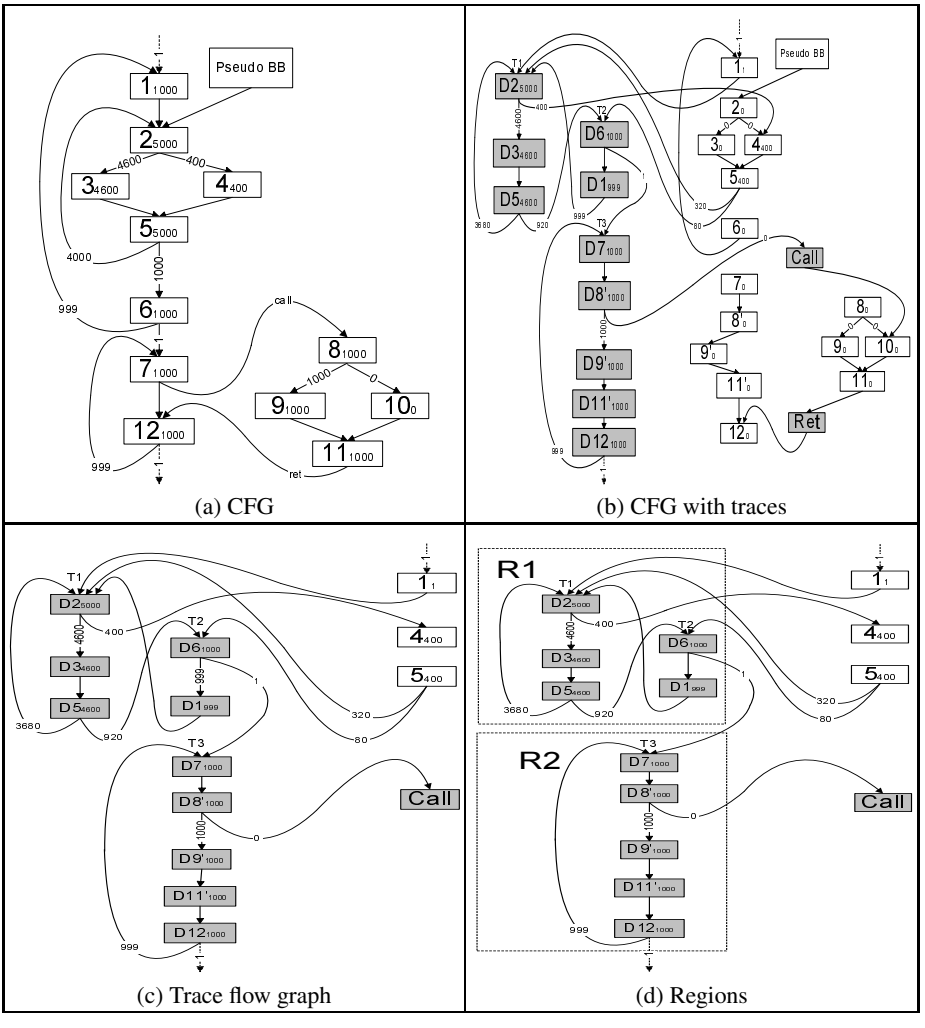
BB_THRESHOLD is a tunable parameter introduced to define BB_MIN and is set to be 5% for Mediabench programs. Depending on the application domains under consideration, appropriate threshold values need to be empirically determined. Unlike [12], BB_MIN can vary from program to program, allowing the traces to be identified and constructed in a program-dependent manner. Once a trace header $h$ is found (line 7), the **while** loop in line 8 grows the trace from $h$ by adding more and more blocks to the trace. The trace always grows from its last block along its hottest outgoing edge (i.e., branch). Let $s$ be the successor block along this edge. The trace is terminated if $s$ is the pseudo block, a trace header or the exit block of the CFG for the program. The trace is also terminated if $s$ is not hot (i.e., *Hot*($s$) does not hold) or $s.freq < b.freq \times$ BB_PROB (i.e., the execution frequency of $s$ has dropped below BB_PROB (%) of that of the trace header $b$). In line 11, a trace that is identified in line 10 will be duplicated with the execution frequencies of all affected blocks and edges being updated appropriately.

In line 12, UpdateHeaderList is called to do two things. First, some blocks in *headerlist* that are no longer hot are removed (line 14). This can happen since part of its execution frequency may have been allocated to its duplicate in a hot trace. Second, in lines 15 – 17, the successor blocks $s$ of every trace exit are examined. If $s$ is not already in a trace, we add $s$ to *headerlist* if it is hot, i.e., when $s.freq \geqslant$ BB_MIN $\times$ BB_PROB (even if $s.freq <$ BB_MIN may hold). Unlike [12], this ensures that both branches of an if statement are included in traces if both are parts of frequently executed paths.

We have also improved [12] by using a profile-guided devirtualisation technique to reduce the number of unknown indirect jumps in virtual call sites. In the case of a virtual call site, `alto` may represent all possible function invocations as unknown indirect jumps. Based on profiling information, our profile-guided devirtualisation pass devirtualises the hot functions invoked at each virtual call site by means of method test [4]. This involves replacing the indirect jump to a hot function by a direct jump guided by a test on the address of the function.

Our illustrating example is given in Figure 3(a). For a block identified by $n_f$, $n$ denotes its block number and $f$ its execution frequency. The number drawn on an edge $(x, y)$ represents its execution frequency; the number is omitted if the edge is the only outgoing edge of $x$. In Figure 3(a), the edge (7,12) introduced by `alto` serves to indicate that block 12 will be executed after the call made in block 7. The edges of this kind are ignored during trace generation. Running our algorithm over the example given in Figure 3(a) produces the modified CFG shown in Figure 3(b). There are a total of three traces generated. They are highlighted in gray boxes, where the trace D7-D8′-D9′-D11′-D12 crosses the boundaries of the two functions in the example.

In Figure 3(a), $(7, 8)$ is a call edge, which is part of the trace denoted by $T_3$. In this case, we rely on the procedure InlineCriticalPaths available in `alto` [15] to inline a frequently executed subgraph rooted at the entry block of the callee. Afterwards, our al-




(a) CFG (b) CFG with traces

(c) Trace flow graph (d) Regions

**Fig. 3.** An example CFG

gorithm will continue to grow the trace on the inlined subgraph as usual. In Figure 3(b), the blocks 7, 8, 9, $8'$, $9'$ and $11'$ are dead, which will eventually be removed.

## 3 Trace-Based Leakage Optimisation

In Section 3.1, we give an algorithm for clustering the traces into optimisation regions, the units of our energy-oriented optimisations. In Section 3.2, we describe the architectural features required for supporting our leakage optimisation. Section 3.3 presents our trace-based optimisation for reducing the data cache leakage energy.

### 3.1 From Traces to Optimisation Regions

The hot traces constructed by GenTrace given in Figure 2 are clustered into the so-called optimisation regions, which are the units of energy-oriented optimisations. Given a region, we will reduce the leakage energy of a data cache by turning off the cache at the entries of the region and turning it back on again at its exits. Since what are inside a region are hot traces, its entries and exits are less frequently executed than the blocks/edges inside. However, the switching on/off activities on these insertion points, if performed too frequently, can still consume significant CPU cycles and dynamic energy. To allow the tradeoffs between performance and energy savings to be made, the granularity of optimisation regions can be tuned.

The formation of optimisation regions relies on a so-called trace flow graph, which is defined below and illustrated in Figure 3(c) using our running example. In addition, the concept of trace flow graph is also used in our two optimisations.

**Definition 1.** *A control flow edge $(x, y)$ in the CFG of a program 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 in addition, $y$ is a trace header.*

**Definition 2.** *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 3(b) is shown in Figure 3(c), 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 $(\mathrm{D1}, \mathrm{D2})$, $(\mathrm{D5}, \mathrm{D2})$, $(\mathrm{D5}, \mathrm{D6})$, $(\mathrm{D6}, \mathrm{D7})$ and $(\mathrm{D12}, \mathrm{D7})$ are trace link edges. Note that $(\mathrm{D5}, \mathrm{D2})$ ($(\mathrm{D12}, \mathrm{D7})$) is a trace link edge for the trace $T_1$ ($T_3$) itself.

After the traces have been constructed, the optimisation regions are formed by calling FindRegions. This procedure expects two arguments to be passed in: *TFG* represents the trace flow graph of a given program and *Affinity* is a value ranging in $[0, 1]$. Essentially, an optimisation 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 depending on the value of *Affinity* so that we can tune the granularity of optimisation regions formed. If *Affinity* $= 0$ (i.e., a small positive number close to 0, in practice), then all regions are singleton traces. Such a setting is the most aggressive in turning off unused or infrequently used hardware components (e.g., cache) in a region. If *Affinity* $= 1$, then every 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 power-aware instructions inserted. Varying the value of *Affinity* allows tradeoffs to be made between energy savings and performance.

Figure 3(d) depicts the two regions formed for the program shown in Figure 3(c) with *Affinity* $= 1/1.2$ under the assumption that $\mathsf{BB\_MIN} = 1000$.

### 3.2 Architecture Support

The leakage power of a CMOS circuit is directly proportional to the product of the power supply voltage ($\mathrm{V_{DD}}$) and the leakage current in a CMOS transistor. Circuit

```
1 PROCEDURE FindRegions(TFG, Affinity)
2 Let L be the set of all trace link edges e in TFG such that e.freq < (1/Affinity − 1) * BB_MIN
3 TFG' = TFG with all edges in L removed
4 return (set of all connected subgraphs in TFG')
```

**Fig. 4.** An algorithm for forming regions

techniques such as power gating (SG), input vector control (IVC) and dynamic voltage scaling (DVS) [2,6] can reduce the leakage power by reducing the supply voltage and/or leakage current. To support our optimisations, we assume the availability of on and off instructions in the underlying instruction set architecture (ISA).

Following [19], we use the same state-preserving leakage control mechanism as proposed in [6], which can preserve the contents of a cache line when the line is put into a low leakage mode. Thus, the cache behaviour of a program is not affected.

The execution of an on (off) instruction causes all the cache lines in the cache to be placed in a normal (leakage-saving) state. Whenever a cache line is accessed, if it is in the leakage-saving state, the normal state will be restored first for the cache line before the access is executed. The execution of an on (off) instruction with respect to an cache line that is in the normal (leakage-saving) state has no effect on the leakage status of the cache line. The latencies and dynamic energy overheads of on/off instructions depend on the exact implementation mechanism.

### 3.3 Leakage Optimisation for Data Caches

Given an optimisation region, all the cache lines in the cache are "turned off", i.e., placed in the low-leakage mode at its entries and "turned on", i.e., placed in the normal mode at its exits. A cache line accessed in a region, once restated to the normal mode, will remain so until the region has been executed.

Figure 5 gives our algorithm, CacheOpt, for reducing the data cache leakage energy. In lines 3 and 4, we identify the traces and then form the optimisation regions. In line 5, we call InsertOnOffInsts to insert the required on/off instructions at the entries and exits of every optimisation region straightforwardly. In lines 8 – 9, we insert one single "off instruction" on every trace entry edge. In lines 10 – 11, we insert one single "on instruction" on every trace exit edge. In lines 12 – 14, we find every trace link edge $(x, y)$ such that $x$ and $y$ are two distinct regions, in which case $x.region \neq y.region$. Every such an edge serves as a exit edge of the region $x$ and an entry edge of the region $y$. Therefore, an "off instruction" is inserted on the edge. Note that an "on instruction" needs not be inserted redundantly before the off instruction on the same edge.

Our algorithm allows the granularity of optimisation regions to be adjusted by varying the tunable parameter AFFINITY. If the regions are large enough, the performance and dynamic energy penalties due to switching on/off activities will be insignificant but the opportunities for leakage reduction are also small. In general, the larger a region is, the larger the number of cache lines there will be in the normal mode and the smaller the leakage energy savings will be in the region. Therefore, the regions can be tuned to

```
1  #DEFINE AFFINITY = a value in [0, 1]
2  PROCEDURE CacheOpt()
3  Build the TFG (Definition 2)
4  SetofRegs = FindRegions(TFG, AFFINITY)
5  InsertOnOffInsts
6  PROCEDURE InsertOnOffInsts()
7  Insert one "on inst" at entry to main
8  for every trace entry edge (x, y)
9      Insert one "off inst"
10 for every trace exit edge (x, y)
11     Insert one "on inst"
12 for every trace link edge (x, y)
13     if x.region ≠ y.region
14         Insert one "off inst"
```

**Fig. 5.** A leakage optimisation for data caches

make tradeoffs between the leakage energy savings and associated overheads (including dynamic energy and execution time penalties).

**Example.** As in before, we assume that BB_MIN=1000, BB_PROB=50% and AFFINITY $= 1/1.2$. Consider our running example given in Figure 3(a). In lines 3 – 4, the trace flow graph and the optimisation regions found are given in Figures 3(c) and 3(d), respectively. As a result, the on and off instructions are inserted as shown in Figure 6.

## 4 Experimental Results

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

We use 15 benchmarks from the Media benchmark suite. All benchmarks are compiled using `DEC C 5.6-075` at "O2" on an Alpha 21264-based system. Similar trends in our results are observed at "O3" or under `gcc` with varying optimisation levels. There are so-called "second data sets" for 12 out of the 15 benchmarks available in the Mediabench web site. The exceptions are `pgpencrypt`, `pgpdecrypt` and `mesa`. For each benchmark, the profiling information is collected using the so-called "second data set" if it exists and the data set that comes with the benchmark otherwise. All benchmarks are simulated using the data sets that come with these benchmarks.

We consider a superscalar out-of-order architecture consisting of two integer multipliers, four integer ALUs for non-multiplication integer operations, one floating point multiplier and four floating point adders. Such an architecture is chosen to match the target architecture of `alto`, in which our trace-based framework is implemented. We use sim-outorder, an out-of-order cycle-level simulator from SimpleScalar. The simulations for all the benchmarks are run to completion.

In order to make our presentation precise, we use $P_{\text{alto}}$ to denote the binary from `alto` and $P_{\text{opt}}$ to denote the binary generated after CacheOpt has been applied.

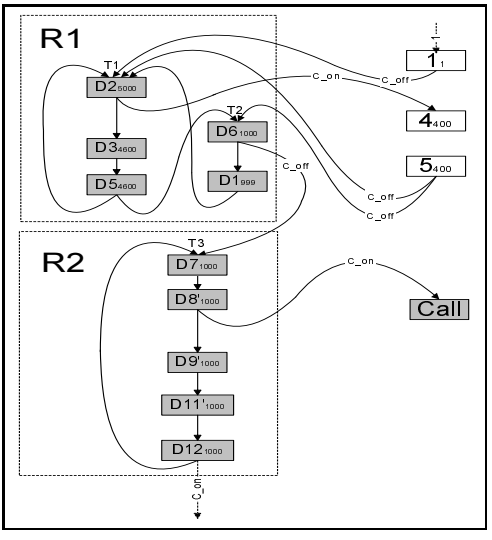



**Fig. 6.** The result of applying CacheOpt to the running example given in Figure 3. All tunable parameters used are defined in Section 3.3.

### 4.1 Trace Generation: GenTrace

The three metrics are used: (1) the trace accuracy measured as the cycles spent in the traces, (2) the code size increase due to the duplication of the traces, and (3) the performance degradation due to the introduction of the traces.

**Table 1.** Five settings for BB_THRESHOLD and BB_PROB

| Configuration | BB_THRESHOLD | BB_PROB |
|---|---|---|
| CONFIG1 | 3% | 50% |
| CONFIG2 | 5% | 25% |
| DEFAULT | 5% | 50% |
| CONFIG3 | 5% | 75% |
| CONFIG4 | 8% | 50% |

We evaluate GenTrace below using the five configurations listed in Table 1, where DEFAULT is the default setting. The trace accuracies are over 80% for all benchmarks under all five configurations. The only exception is `djpeg` for which an accuracy of 49.12% is obtained in CONFIG1. In this special case, a threshold of BB_PROB $= 3\%$ results in BB_MIN=703, which is too large to capture all frequently executed paths in the benchmark. The static instruction count increases range from 0.12% in nearly all five configurations for both `rawcaudio` and `rawdaudio` to 6.87% in CONFIG4 for `cjpeg`. The performance changes for all the benchmarks are very encouraging. Out of the 15 benchmarks used, `pgpdecrypt` and `g721encode` have small positive or

**Table 2.** Cache parameters taken from [19]

| Parameter | Value |
|---|---|
| Feature size | 0.07 micron |
| Supply voltage | 1.0 V |
| L1 I-cache | 16 KB, direct-mapped |
| L1 I-cache latency | 1 cycle |
| L1 D-cache | 16 KB, 4-way |
| L1 D-cache latency | 1 cycle |
| Unified L2 cache | 512KB, 4-way |
| L2 cache latency | 10 cycles |
| Memory latency | 100 cycles |
| Clock speed | 1 GHz |
| L1 cache line size | 32 bytes |
| L2 cache line size | 64 bytes |
| L1 cache line leakage energy | 0.33 pJ/cycle |
| L1 deactive mode cache line leakage energy | 0.01 pJ/cycle |
| L1 state-transition (dynamic) energy | 2.4 pJ/transition |
| L1 state-transition latency from deactive mode | 1 cycle |
| L1 dynamic energy per access | 0.11 nJ |
| L2 dynamic energy per access | 0.58 nJ |

negative speedups configurations under all five configurations, `toast` and `untoast` run between 0.04% to 1.58% slower under all five configurations, and the remaining 11 programs run faster under all five configurations. These performance variations appear to be attributed to the profile-guided code layout pass invoked in the code generation module of `alto` as shown in Figure 1. Our results show that GenTrace is capable of identifying the most frequently executed paths in a program and the associated costs for duplicating these paths as the hot traces in the program are small (relative to the achieved energy savings to be discussed shortly).

### 4.2 Leakage Optimisation for Data Caches

We will use the DEFAULT configuration to evaluate our data cache leakage optimisation: BB_THRESHOLD $= 5\%$, BB_PROB $= 50\%$ and AFFINITY takes four values: 1, 1/1.05, 1/1.5 and 0. We adopt the cache configuration and energy numbers listed in Table 2, which is taken entirely from [19]. The cache state-preserving leakage control mechanism used is from [6]. According to [19], the energy numbers were obtained by circuit simulation for the 0.07 micron process.

Figure 7 depicts the data cache leakage energy savings achieved by CacheOpt. The percentage leakage reduction in a program $P_{\text{opt}}$ is given by:

$$\text{cache\_saving} = \frac{O_{\text{static}} - C_{\text{dynamic}} - C_{\text{static}}}{O_{\text{static}}}$$

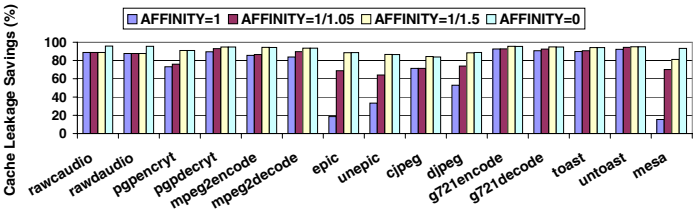



**Fig. 7.** Percentage cache leakage energy reductions of $P_{\text{opt}}$ relative to $P_{\text{alto}}$

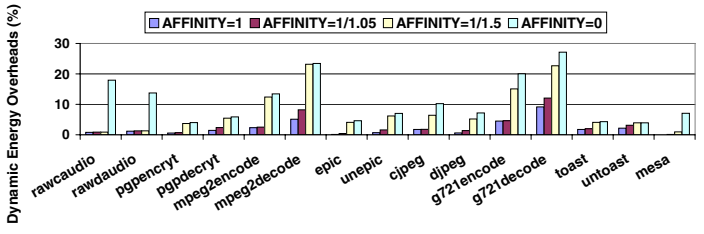



**Fig. 8.** Dynamic energy overheads measured as $\frac{C_{\text{dynamic}}}{C_{\text{static}}+C_{\text{dynamic}}}$

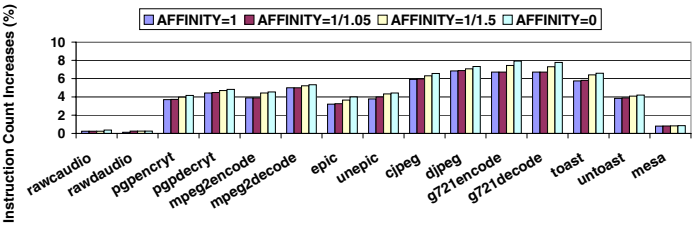



**Fig. 9.** Code expansion of $P_{\text{opt}}$ relative to $P_{\text{alto}}$

where $O_{\text{static}}$ denotes the amount of leakage energy consumed in $P_{\text{alto}}$ before the optimisation, $C_{\text{static}}$ the amount of leakage energy consumed in the optimised $P_{\text{opt}}$ and $C_{\text{dynamic}}$ the dynamic energy overhead due to the switching on/off activities introduced in $P_{\text{opt}}$. As shown in Figure 7, the leakage energy savings are obtained in all benchmarks at all four AFFINITY values. In addition, they are progressively non-worse as the granularity of optimisation regions (i.e., AFFINITY) decreases. When AFFINITY $= 1$, the leakage reductions range from 18.59% for `epic` to 92.59% for `g721encode`. In the other extreme when AFFINITY $= 0$, the leakage reductions are more impressive, ranging from 83.87% for `cjpeg` to 95.74% for `rawcaudio`.

In the Mediabench benchmarks, a small set of data are typically active at a given period of time. As a result, reducing the granularity of optimisation regions tends to increase the total leakage energy saved. While smaller regions lead to higher on/off switching activities, i.e., higher dynamic energy consumption, as illustrated in Figure 8, these overheads are more than or equally outweighed by the leakage energy savings

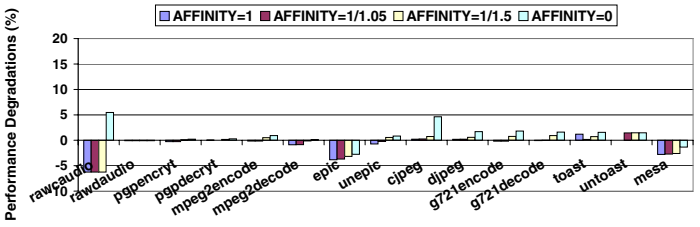



**Fig. 10.** Performance changes of $P_{\text{opt}}$ relative to $P_{\text{alto}}$

achieved at all the four AFFINITY values used. This phenomenon is more pronounced in `pgpencrypt`, `epic`, `unepic`, `cjpeg`, `djpeg` and `mesa`. In the other nine benchmarks, the largest optimisation regions obtained when AFFINITY $= 1$ are small, resulting in already at least 83.87% leakage reduction in each case. So any further leakage savings from using smaller regions are relatively insignificant.

The impact of CacheOpt on code size and performance is illustrated in Figures 9 and 10. In both cases, the cost increases are relatively small.

## 5 Related Work

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 [13], register allocation [7] and instruction scheduling [11]. Loop transformations such as loop tiling can reduce the dynamic energy spent on cache by reducing the cache misses in the program [9]. 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, [16] explore DVS as a means of improving the dynamic energy consumption of a program without increasing its execution time. [18] analyse and evaluate the opportunities and limits of compile-time DVS scheduling.

The on-chip caches are one of the hardware components for leakage reduction since they contain a significant fraction of the transistors in a microprocessor. Flautner *et al* [6] present architectural techniques for reducing the leakage energy of a data cache by periodically putting cache lines into a low-power mode. Motivated by this work, Zhang *et al* [19] describes a loop-based, compiler-directed solution. Essentially, the innermost loops are taken as optimisation regions. Given an innermost loop, all the cache lines are placed in a low-leakage mode at the beginning of the loop and restored to their normal mode at its exits. His experimental results over benchmarks show that this software solution can be competitive with the hardware-based solution [6].

In this work, we present a trace-based approach to reducing data cache leakage energy at link time. Rather than innermost loops, our units of optimisations are the regions

constructed from the hot traces. The advantages of using traces are stated earlier. The traces are inherently inter-procedural, spanning both user and library functions (which may contain assembly code). In addition, the frequently executed paths formed by recursive calls are recognisable as traces but not as loops.

There are a number of static or dynamic binary translation systems around [1,3,17]. These systems aim at improving performance or otherwise achieving portability. However, we are the first to investigate the effectiveness of a trace-based, static binary translation framework in supporting energy-oriented optimisations for embedded applications. Working on binaries at link time (i.e., statically) dispenses with an expensive run-time system that would otherwise be required.

Traces are not new. Trace scheduling [5] 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 [14] and regions [8]. 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. This work demonstrates that the traces also represent a suitable framework to support energy-oriented optimisations.

Our trace generation algorithm identifies the hot traces across procedural boundaries at link time based on an inter-procedural 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 [5,8,14] or cause less trouble when the traces are constructed at run time [1].

## 6 Conclusion

This work investigates for the first time the effectiveness of conducting energy-oriented optimisations for data caches in a traced-based compilation framework at link time. We present a simple yet effective algorithm for identifying and constructing the hot traces in a binary program at link time. We also introduce a trace-based optimisation for reducing leakage energy for data caches. The optimisation is simple since traces allow the optimisation regions and on/off insertion points to be identified easily and also effective since significant leakage energy reductions can be obtained for benchmarks at small performance degradations and code size expansions.

## Acknowledgements

This work is supported in part by ARC Discovery grants DP0211793 and DP0452623.

## 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. A. Chandrakasan, W. J. Bowhill, and F. Fox. *Design of High-Performance Microprocessor Circuits*. IEEE Press, 2001.
3. 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.
4. D. Detlefs and O. Agesen. Inlining of virtual methods. In *13th European Conference on Object-Oriented Programming (ECOOP'99)*, pages 258–278, 1999.
5. J. Fisher. Trace scheduling: a technique for global microcode compaction. In *IEEE Transactions on Computers*, pages 478–490, 1981.
6. K. Flautner, N. S. Kim, S. Martin, D. Blaauw, and T. Mudge. Drowsy caches: simple techniques for reducing leakage power. In *29th annual international symposium on Computer architecture*, pages 148–157. IEEE Computer Society, 2002.
7. 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.
8. 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.
9. 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.
10. N. S. Kim, K. Flautner, D. Blaauw, and T. Mudge. Drowsy instruction caches: leakage power reduction using dynamic voltage scaling and cache sub-bank prediction. In *Proceedings of the 35th annual ACM/IEEE international symposium on Microarchitecture*, pages 219–230. IEEE Computer Society Press, 2002.
11. 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.
12. L. Li and J. Xue. A trace-based binary compilation framework for energy-aware computing. In *LCTES '04: Proceedings of the 2004 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools*, pages 95–106. ACM Press, 2004.
13. 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.
14. 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.
15. R. Muth. *ALTO: A Platform for Object Code Modification*. PhD thesis, The University of Arizona, 1999.
16. 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.
17. 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.
18. 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.
19. W. Zhang. Compiler-directed data cache leakage reduction. In *IEEE Computer Society Annual Symposium on VLSI Emerging Trends in VLSI Systems Design*. IEEE Computer Society, 2004.