Power Aware WCET Analysis

Thesis

Presented in Partial Fulfillment of the Requirements for the Degree
Master of Science in the Graduate School of The Ohio State
University

By

Wenlei Bao, M.S.
Graduate Program in Electrical and Computer Engineering

The Ohio State University

2014

Master’s Examination Committee:

Dr. Fusun Ozguner, Advisor
Dr. P. Sadayappan
© Copyright by

Wenlei Bao

2014
Abstract

Worst case execution time (WCET) analysis is used to verify that real-time tasks on systems can be executed without violating any timing constraints. Power consumption is not considered in most of the WCET research work. However, real-time embedded systems have limited power resource and leakage power cost by cache could take up to 40% of the total power consumption. Because of the data reuse in programs, not all of the cache are needed during the execution. However, traditional WCET analysis, which obtain WCET bound by assuming all cache can be utilized, results in a higher power dissipation.

In this thesis, we propose a Power-aware WCET analysis framework (PWCET) to improve energy efficiency by reducing cache size with no penalty on WCET result. Compiler analysis based useful cache calculation algorithm is used to determine non-useful cache size to be turned off to save power. Experiments of benchmarks demonstrate the effectiveness of the PWCET framework by obtaining considerable power saving without any effect on WCET bounds.
This is dedicated to my parents.
Acknowledgments

First of all, I would like to express my gratitude to my advisor Dr. Fusun Ozguner for her advice and support on my work. Her insightful comments and guidance have inspired me to do research high performance computing and embedded system areas.

My gratitude also goes to Dr. Saday for his discussion, comments and help on my research topics.

No words can express my sincere gratitude and appreciation to my parents for their support in my life.
Vita

2010 ........................................B.E. Harbin Institute of Technology

2012 ........................................M.S. Harbin Institute of Technology

2012-present ..........................Graduate Teaching and Research Associate, The Ohio State University

Fields of Study

Major Field: Electrical and Computer Engineering
# Table of Contents

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>Introduction</td>
<td>1</td>
</tr>
<tr>
<td>2.</td>
<td>Power-aware WCET Analysis Framework</td>
<td>4</td>
</tr>
<tr>
<td>2.1</td>
<td>Background</td>
<td>4</td>
</tr>
<tr>
<td>2.1.1</td>
<td>Notation</td>
<td>5</td>
</tr>
<tr>
<td>2.1.2</td>
<td>Polyhedral Reuse Distance Vectors</td>
<td>6</td>
</tr>
<tr>
<td>2.1.3</td>
<td>Working Set and its Calculation</td>
<td>6</td>
</tr>
<tr>
<td>2.1.4</td>
<td>WCET Analysis</td>
<td>8</td>
</tr>
<tr>
<td>2.2</td>
<td>Determination of Useful Cache Size in PWCET</td>
<td>9</td>
</tr>
<tr>
<td>2.3</td>
<td>Power-aware WCET Analysis</td>
<td>10</td>
</tr>
<tr>
<td>2.4</td>
<td>Multi-core Analysis</td>
<td>12</td>
</tr>
</tbody>
</table>
## List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1 PWCET vs. Traditional Static Timing analysis</td>
<td>18</td>
</tr>
<tr>
<td>3.2 PWCET for Multi-core Analysis</td>
<td>24</td>
</tr>
</tbody>
</table>
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>An example of an affine computation</td>
<td>4</td>
</tr>
<tr>
<td>2.2</td>
<td>Static WCET framework</td>
<td>9</td>
</tr>
<tr>
<td>2.3</td>
<td>PWCET framework</td>
<td>13</td>
</tr>
<tr>
<td>2.4</td>
<td>Matrix multiplication using OpenMP</td>
<td>14</td>
</tr>
<tr>
<td>3.1</td>
<td>Useful Cache Size vs. Problem Size</td>
<td>20</td>
</tr>
<tr>
<td>3.2</td>
<td>WCET penalty and Power Saving vs. Useful Cache Size</td>
<td>22</td>
</tr>
</tbody>
</table>
Chapter 1: Introduction

Power consumption for real-time embedded systems such as wireless sensor system is an important problem since the power resource is limited. So improving system energy efficiency becomes a crucial topic.

With the increasing of cache size on the processor die area (more than 40%) [3] and leakage power consumption [15], the large cache results in a considerable fraction of the total power consumption. Moreover, not all of the cache can be fully utilized because of the inherent data reuse in programs. In other words, unused cache can be turned off to reduce power consumption without affecting the performance.

The topic of cache-based power saving techniques have already been studied by many researchers. Approaches proposed in [21],[30],[23] and [10] have a common scheme which is to first observe the cache miss-rate and then the resize cache based on observations. The problem of this dynamic scheme is that the stored data have to be saved before that part of the cache can be turned off.

To improve the energy efficiency of real-time systems, the system stringent timing constraints have to be satisfied since any violation could generate catastrophic effects [8]. Therefore, worst case execution time (WCET) analysis is needed to validate such systems by obtaining the upper bounds of execution time. It can then
be determined whether computational tasks can execute on the system based on the upper bounds.

However, most research on WCET analysis focuses on processor behavior modeling to derive tight WCET bounds. There is not much WCET research work that considers system energy efficiency. Shared level-2 cache analysis is incorporated into hardware modeling in WCET analysis in [29]. Authors in [16] takes bus access modeling into consideration to obtain timing bounds for architectures with bus interconnections. Reference [9] combines shared cache analysis with bus access modeling. To the best of our knowledge, only Kiran et al’s work [22] takes care of power consumption issues in static timing analysis. Their proposed frequency-aware static timing analysis is based on the DVS technique, which is not fit for processors without support of DVS. Moreover, WCET analysis has no concern of non-useful cache during execution, which results in a higher system power consumption.

Therefore, we propose a novel power-aware WCET analysis technique to improve system energy efficiency and validate real-time tasks at the same time. we made following contributions in this thesis: 1) An extended algorithm based on compiler analysis to determine useful cache size for WCET analysis is presented. 2) A Power-aware WCET framework to improve system energy efficiency without penalty in WCET bound is proposed. 3) The transition points of effective cache with different WCET penalties are generated for several benchmarks, demonstrating that PWCET can be used efficiently by systems with different preferences to tradeoff between power savings and WCET penalty. 4) Useful cache calculation for parallel programs is proposed, and analysis for multi-core platform is also presented.

The rest of this thesis is organized as follows.
Chapter 2 introduces background knowledge first, and then describes the proposed Power-aware WCET Framework including the useful cache calculation algorithm.

Chapter 3 describes the experiments in detail. It contains the tools and benchmarks used in the experiments. Results are presented and discussed in this chapter.

Chapter 4 presents the conclusions of the thesis.
Chapter 2: Power-aware WCET Analysis Framework

This chapter describes Power-aware WCET analysis. The PWCET analysis is based on a compiler analysis to determine the useful cache size. PWCET framework and its components are also described in detail in this section. Before presenting the PWCET and the framework, we include the background information used in useful cache analysis and WCET analysis. These basic notions and definitions are necessary in the following description.

2.1 Background

Notation and working set described in this section are used in useful cache calculation algorithm of the PWCET analysis.

```c
for (t = 0, t <= tsteps; t++)
{
    for (i = 0; i <= n; i++)
        for (j = 0; j <= n; j++)
            S: access A[i][j], A[i-1][j];
}
```

Figure 2.1: An example of an affine computation
2.1.1 Notation

Basic notations used are the same with those used by LaMielle [18] including sets, relations and the apply operation.

1) Sets: The definition of a set $s$ is:

\[ s = \{[i_1, ..., i_m] : c_1 \land ... \land c_n \} \]

where each $i_k$ is an iterator and each $c_j$ is a loop constraint. The sets are used to express iteration space of loops. The iteration space of loop in Figure 2.1 can be denoted as the set $I$:

\[ I = \{S[t, i, j] : (0 \leq t \leq T) \land (0 \leq i \leq n) \land (1 \leq j \leq n)\} \]

2) Relations: The definition of a relation $r$ is:

\[ r = \{[i_1, ..., i_m] \mapsto [j_1, ..., j_n] : c_1 \land ... \land c_o \} \]

where each $i_x$ is an input iterator and each $i_y$ is an output iterator. Relations can be used to describe the array references. For example, the access of array $A[i][j]$ and $A[i - 1][j]$ of statement $S$ in Figure 2.1 can be represented as following relations:

\[ r_1 = \{S[t, i, j] \mapsto A[i_1, j_1] : (i_1 = i) \land (j_1 = j)\} \]
\[ r_2 = \{S[t, i, j] \mapsto A[i_1, j_1] : (i_1 = i - 1) \land (j_1 = j)\} \]

3) Apply Operation: The definition of apply operations is:

\[ (\bar{x} \in s') \iff (\exists \bar{y} \in s \land (\bar{y} \mapsto \bar{x}) \in r) \]

where $s'$ is a new set produced by apply relation $r$ to set $s$ which can be denoted as $s' = r(s)$. 

5
The apply operation is used to determine the data footprints of array referenced in the loop. For example, the data footprints are \( r_1(I), r_2(I) \) for the accesses of array \( A[i][j] \) in Figure 2.1.

2.1.2 Polyhedral Reuse Distance Vectors

The iteration space of an \( n \) depth nested loop is viewed as a convex integer polyhedron in the Polyhedral model [28, 11]. Iteration vector \( \vec{v} = (v_1, v_2, ..., v_n) \) is used to specify a particular iteration, where \( v_i \) is one of the iterators of the loop. Distance vector is the difference between two iteration vectors.

When the same data is accessed in these two iterations, the distance vector is used to denote this reuse and becomes the reuse vector. In Figure 2.1, iteration vectors \( (t, i, j) \) and \( (t + 1, i, j) \) reference the same element of \( A[i][j] \), so one reuse vector is \( (1, 0, 0) \).

2.1.3 Working Set and its Calculation

To realize the data reuse in cache denoted by the reuse vector, all the data referenced in a reuse vector should be kept in the cache. These data elements are called the working set [24]. Basically, the useful cache analysis is to determine the size of the working set.

Here we only show the working set calculation for reuse vector with one nonzero element. The calculation for the general reuse vector can be constructed based on this.

Let set \( S \) denote an iteration space with \( n \) depth nested loops:

\[
S = \{ [x_1, x_2, ..., x_n] : c_1 \land c_2 \land ... \land c_n \}
\]
with reuse vector $\vec{v}_k = (0, ..., v_k, ...0)$, where $x_1, x_2, ..., x_n$ are loop iterators and $c_1, c_2, ..., c_n$ are constraints. Note that the reuse vector should have at least one non-zero element. Otherwise, it is a trivial reuse. The iteration space expressed by the reuse vector can be denoted as:

$$S_{\vec{v}_k} = \{[x_1, ...x_k, ...x_n] : (1 \leq x_1 \leq 1) \land ... \land (1 \leq x_{k-1} \leq 1)$$

$$\land (1 \leq x_k \leq |v_k|) \land c_{k+1}... \land c_n \}$$

where the outer loop iterations, $x_1$ to $x_{k-1}$, are set to 1 to remove them from the consideration for working set, while inner loop iterations of $i_k$ are set to go through the entire loop length. The loop bound for iteration $x_k$ is set to $|v_k|$.

Perform apply operation for each relation $r$ to corresponding iterations space $s_{\vec{v}_k}$ following by union will provide the data footprint of array references, which is the working set of array $A$ for the reuse vector $\vec{v}_k$.

$$WS_{\vec{v}_k} = \{r_1(S_{\vec{v}_k}) \cup r_2(S_{\vec{v}_k}) \cup ... \cup r_n(S_{\vec{v}_k}) \}$$

The cache size for a working set depends on the data layout. We assume the array elements in working set start at address 0 and are then mapped to continuous memory locations. Consider a row major ordering 2-dim array with $N^2$ elements. Each element takes $E$ bytes and the cache line is $B$ bytes. Then the linear addresses and cache line size can be obtained by the following relations.

$$r_{linear} = \{[i, j] \mapsto [N \times E \times i] + E \times j \}$$

$$r_{line} = \{[i] \mapsto [i/B]\}$$

Applying relation $r_{line}$ to the linearized working set will provide the number of cache lines:

$$NW_{S_{\vec{v}_k}} = \{r_{line}(r_{linear}(WS_{\vec{v}_k}))\}$$
The details of working set calculation can refer to previous work [24].

### 2.1.4 WCET Analysis

There are basically two classes of methods in WCET analysis [27]: measurement-based method (dynamic method) and static method. The measurement-based method calculates the WCET bound by measuring the execution time of a program on hardware or a simulator for a subset of all possible inputs. It is unsafe to consider only a subset of inputs because there is no guarantee the subset inputs will generate the worst case execution time. Therefore, the measurement-based method, which underestimates the WCET, is often used in industry to get a first rough estimate of the execution time [17].

The static method guarantees to generate a safe timing bound by considering all possible executions of the program. Abstractions are made in this process to make the analysis feasible. It consists of 3 analysis phases: control flow analysis, architecture modeling and WCET calculation. Control flow analysis constructs and analyzes the control flow graph based on the program. Architecture modeling models the processor features including caches and pipelines. Finally, the WCET bound is calculated in WCET calculation phase by performing integer linear programming (ILP) or implicit path numeration (IPET) [20] on constraints generated in previous phases. This process can be seen in Figure 2.2.

There are many WCET analysis tools developed based on static method, commercial tools such as aiT [12], research tools including Chronos [19], OTAWA [4].
2.2 Determination of Useful Cache Size in PWCET

The PWCET analysis to improve system energy efficiency based on useful cache analysis is presented in this section. Multi-core analysis for parallel programs is also described.

The useful cache calculation algorithm [24] takes affine computations in programs as inputs. In affine computations the array references and iterations are affine functions of loop iterators and program parameters. These programs are of a vital category of programs according to a study [5]. Other inputs to the useful cache calculation algorithm are the maximum cache size and problem size.

The reuse distance vectors of affine computations are obtained by performing polyhedral dependence analysis [6]. The number of working set cache lines is calculated for each reuse vector based on techniques described in the previous section. The different working sets are then sorted in increasing order of their sizes and cache lines
are allotted to arrays starting with the smallest working set. Finally, the total useful cache size is returned together with per array useful cache lines.

We extend the useful cache calculation algorithm in PWCET analysis. Cache lines used by codes other than affine loop nests in programs are determined conservatively. For example, a double precision floating point variable declaration is assumed to take a single line in cache. And, the useful cache size of several disjoint affine computations is calculated by union to find the largest working set.

The algorithm is the extended algorithm to determine the useful cache size for a given program in PWCET. Inputs to the algorithm are problem sizes and maximum cache lines. Affine loop nests are determined at the beginning to be analyzed first. For affine computations, useful cache lines of each affine loop-nest are calculated by union to find total affine useful cache size. Then non-affine computations are analyzed to estimate their useful cache size. The total useful cache size is calculated by combining these two different cases, and returned as the output.

2.3 Power-aware WCET Analysis

The PWCET framework and its components are shown in Figure 2.3. The framework takes the source program as input. The program is then translated to the control flow graph by the compiler. The generated control flow graph is needed for both hardware modeling and control flow analysis.

The hardware modeling phase analyzes the control flow graph and predicts the processor behaviors with the goal to derive architectural constraints which will be used in WCET calculation phase. The generated architectural constraints are used to determine the execution time of basic blocks. A basic block is a block of consecutive
Algorithm DeterminUsefulCache

Output: Total Useful Cache Line: $TotalUsefulCacheLines$.

1. $TotalUsefulCacheLines \leftarrow 0$;
2. $AffineUsefulCacheLines \leftarrow 0$;
3. Affine Computations: $P \leftarrow$ DetermineAffineComputations($T$);
4. for all Affine Nested Loop: $A \in P$ do
5.     $WorkingSetLines \leftarrow \emptyset$;
6.     for all Array: $X \in A$ do
7.         $ReuseVectors \leftarrow$ PolyhedralDependencies($A$);
8.         for all ReuseVector: $V \in ReuseVectors$ do
9.             $NumWorkingSetLines \leftarrow$ Compute_NWSL($V, X, A$);
10.            $AddToWorkingSetLines(X, NumWorkingSetLines)$;
11.         end for
12.     end for
13.     SortWorkingSetLines($WorkingSetLines$);
14.     $PerArrayCacheLines[#X] \leftarrow 0$;
15.     $UsefulCacheLines \leftarrow 0$;
16. for all NumWorkingSetLines: $NWSL \in WorkingSetLines$ do
17.     $X \leftarrow$ GetArray($NWSL$);
18.     if $NWSL \leq (MaxCacheline - PerArrayCacheLines(X)) + NWSL$ then
19.         $UsefulCacheLines \leftarrow (UsefulCacheLines - PerArrayCacheLines(X) + NWSL)$;
20.         $PerArrayCacheLines(X) \leftarrow NWSL$;
21.     end if
22. end for
23. $AffineUsefulCacheLines \leftarrow$ Union($UsefulCacheLines$);
24. end for
25. $NonAffineCacheLines \leftarrow$ DetermineNonAffineUsefulCache($T$);
26. $TotalUsefulCacheLines \leftarrow$ Union($AffineUsefulCacheLines, NonAffineCacheLines$);
27. return $TotalUsefulCacheLines$;
instructions in control flow graph, which can only be entered through the first instruction and exited at the last instruction. *Control flow analysis* is used to derive the flow constraints which contains the path information such as infeasible loops, which are necessary in WCET bound calculation. Besides these constraints, user annotations such as loop iterations are also needed to derive the WCET bound.

In *Useful cache analysis*, our determine-useful-cache algorithm is invoked to calculate the useful cache size based on the compiler analysis as we described above. Useful cache size determined by the algorithm is then used to configure the cache hierarchy when performing *hardware modeling*. Leakage power consumption is estimated by *power analysis* based on the resized cache.

The *WCET calculation* phase, which is based on implicit path enumeration or the integer linear programming technique is performed after obtaining all necessary constraints generated by previous phases. Then the calculated bound is examined to determine whether it is acceptable in the *evaluation* step. The bound, if acceptable, is then outputted as the final WCET bound together with power saving information. Otherwise, PWCET will re-perform the *useful cache analysis* step to resize the useful cache to generate an acceptable WCET bound.

Note that the determine-useful-cache algorithm can also be integrated with the measurement-based methods to form a measurement-based framework to gain improvement in energy efficiency.

### 2.4 Multi-core Analysis

The PWCET framework works also for multi-core platforms. However, the useful cache analysis will be different and the working set size may increase compared to
Figure 2.3: PWCET framework
the sequential case. Let us consider a multi-core system with 2 cores and shared cache and shared memory. We consider OpenMP programs to make use of multi-core processors. For example, in the matrix multiplication code shown in Figure 2.4, the outer loop $i$ is parallel in iteration space $(i, j, k)$, and is executed on 2 processors. Let the iterations on processor 0 be denoted as $i_0$, and iterations on processor be 1 denoted as $i_1$.

For array $B$, the references are not related to iterator $i$, so the working set for array $B$ is not changed from sequential case. However, $i$ corresponds to rows in the references of array $C$, thus $i_0$ and $i_1$ refer to different rows. Thus the working set size for array $C$ is doubled because different data will be held in shared cache at the same time. Similar case for array $A$. The above description applies when the reuse is within or at the parallel loop – reuse vector $(1, 0, 0)$ is at the parallel loop $i$ in this case.

When the reuse is above the parallel loop, for example, a reuse vector $(1, 0, 0, 0)$ for an iteration space $(t, i, j, k)$ where $i$ is parallel, the working set size remains unchanged.

```
#pragma omp parallel for
for (i = 0, t < N; t++)
  for (j = 0; i < N; i++)
    for (k = 0; j < N; j++)
      C[i][j] += A[i][k]*B[k][j];
```

Figure 2.4: Matrix multiplication using OpenMP
We conclude general multi-core analysis as following. If reuse occurs above parallel loop,

$$WS_{\text{parallel}} = WS_{\text{sequential}}$$

If reuse occurs within or at the parallel loop,

$$WS_{\text{parallel}} = P \times WS_{\text{sequential}}$$

where $P$ is the number of processors. Note that this is an overestimation because we can see from the analysis that working set size of $B$ is not changed. If a more accurate analysis is needed, useful cache analysis should be performed separately for each array.
Chapter 3: Experimental Evaluation

In this chapter, we describe the experiments to validate the PWCET framework. The framework is used to derive the WCET bounds, power savings and cache savings for a number of WCET benchmarks over a range of problem sizes.

To conduct the experiments of PWCET framework, aiT [12] is used to perform the static timing analysis to obtain WCET bounds. aiT analyzer, which is a software tool that has been widely used in academia and industry to derive worst case execution time bounds for applications, guarantees that the bounds are safe. MPC603e microprocessor, which has 16-Kbyte data caches with 4-way associativity, is selected as a case study since it is well supported by aiT. The cross-compiling GCC for MPC603e is built on a linux machine and used to translate the source code to executable files since aiT directly analyzes binary executables. Iscc [2], an interface to use barvinok library [25] to count integer points in parametric polytopes, is used to implement the useful cache calculation algorithm.

For each WCET benchmark, both traditional WCET and PWCET analysis are performed. Generated WCET bounds are recorded together with power consumptions obtained by CACTI [1] to estimate the power savings based on the parameters according to the processor.
3.1 Benchmarks

The benchmarks used in the experiments are bsort100, cnt, ludcmp, matmult, tiled-Matmult, minver and ud.

Bsort100 is a bubble sort program that is used to test the basic loop constructs and array handling by sorting 100 integers. Cnt is a program with nested loops to count non-negative numbers in a matrix. Ludcmp is the LU decomposition algorithm for floating point arrays. Matmult is matrix multiplication of two matrices with nested loops and function calls. Minver is a code to perform inversion of floating point matrix. Ud is a calculation of matrixes with 3-level nested loops.

The matmult code is tilable and tiling [7] can provide a better data reuse in cache. The tiled version tiled-Matmult can obtain better data reuse compared to matmult code. So both the tiled and un-tiled version are included as benchmarks in this thesis. The tile size is chosen to get best performance and known as given to the analysis in the experiments. For other benchmarks, tiling is not considered because of feasibility.

All benchmarks except tiled-Matmult are from WCET Benchmarks provide by the Malardalen WCET research group [14]. Among these benchmarks some programs are selected for the WCET Challenge [13, 26].

3.2 Experimental Results

3.2.1 Validation of PWCET framework

The benchmarks and experimental results are listed in Table 3.1. The original data cache size is 512 cache lines with line size 32 bytes. The useful cache size determined by the compiler analysis based algorithm is resized to the value of power of two and shown under the ‘Resized Cache’ column. WCET bounds generated by traditional
Table 3.1: PWCET vs. Traditional Static Timing analysis

<table>
<thead>
<tr>
<th>Benchmarks</th>
<th>Static Analysis</th>
<th>PWCET</th>
<th>Power Saving</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>DCache</td>
<td>WCEC</td>
<td>Power</td>
</tr>
<tr>
<td>bsort100</td>
<td>512</td>
<td>422240</td>
<td>210.35</td>
</tr>
<tr>
<td>cnt</td>
<td>512</td>
<td>12575</td>
<td>210.35</td>
</tr>
<tr>
<td>ludcmp</td>
<td>512</td>
<td>10860</td>
<td>210.35</td>
</tr>
<tr>
<td>matmult</td>
<td>512</td>
<td>541265</td>
<td>210.35</td>
</tr>
<tr>
<td>tiled Matmult</td>
<td>512</td>
<td>614535</td>
<td>210.35</td>
</tr>
<tr>
<td>minver</td>
<td>512</td>
<td>11547</td>
<td>210.35</td>
</tr>
<tr>
<td>ud</td>
<td>512</td>
<td>9468</td>
<td>210.35</td>
</tr>
</tbody>
</table>

static timing analysis with original 16KB cache and PWCET analysis with resized cache are described in cycles under ‘WCET’ columns. The worst case execution cycles remain unchanged for all benchmarks after resizing cache size determined by useful cache analysis.

The leakage power consumption under different cache size are also shown in the table. Leakage power with original cache size is 210.35 mW. Both bsort100 and cnt show the highest 94.71% power savings with resized cache. Benchmark cnt has only spatial reuse but no temporal reuse, therefore it has a small working set. The working set size of bsort100 is also small because it only has one dimension array and the reuse vector for 1-dim array is smaller compared to multi-dim array according to working set analysis.

Other benchmarks including ludcmp, minver and ud show more than 85% savings of leakage power. Matmult shows 73.36% power reduction and the useful cache size of tiled version tiled-Matmult is half of the original cache size with only 49.28% leakage power saving.
The experiments are also conducted with other problem sizes for each of the benchmarks. In 24 out of the 28 cases, the WCET in cycles obtained by PWCET have no increasing compared to traditional WCET analysis. In other words, there is no further overestimation of WCET bounds, but gained much leakage power savings by resizing useful cache based on the complier analysis. In other 4 cases, there exist less than 0.001% worst case execution cycles increase because of conflict misses.

Since power budget is limited especially on embedded systems and leakage power cost is considerable, PWCET analysis, which can obtain tight WCET bounds with resized cache to reduce leakage power consumption, will improve the system energy efficiency compared to the typical power consumption of MPC603e (4W with 300MHz).

### 3.2.2 Relation between Reduced Cache and Problem Size

Figure 3.1 shows the relation between useful cache size and problem size. For each benchmark, useful cache size is calculated by varying problem size from 1 to 1000 with step of 5. The caches that can be turned off in percentage without affecting worst case execution time are shown in the figure according to different problem size.

For Cnt benchmark shown in Figure 3.1 (b), the percentage of turned off cache keeps at 94.71% no matter with problem size. Bsort only contains one dimension array and the working set size is less than problem size N. Hence, all working sets can be hold by cache even for the maximum problem size. Therefore, the percentage of cache that can be turned off decrease steadily with the problem size increases, which is shown in Figure 3.1(a). Similar trend for Ud benchmark except that the percentage of turned off cache decrease to zero when problem size is large enough.
Figure 3.1: Useful Cache Size vs. Problem Size
Other benchmarks, Ludcmp, Matmult and Minver, show that the cache that can be turned off steadily decease at the beginning when the problem size is small compared to the whole cache size. With the increasing of problem size, the useful cache size also increases and the cache that can be turned off decreases to zero. When the useful cache size of the program can no longer be held in cache along with the increasing of problem size, next smaller working set that can fit in cache works again. Thus much of the cache can be turned off again since the new useful cache size based on this smaller working set is much smaller than the whole cache size.

Take Matmult benchmark as an example, the working set increase as $N^2$ initially. However, when it cannot be held in cache with $N$ increases, the next smaller working set to realize reuse is $2N$. Thus, a large percentage of cache can be turned off again after $N = 66$, and that percentage decreases as $N$ increases.

The tiled version program doubles the number of loops compared to the untiled one. Hence, it has larger reuse vectors and working sets. Therefore, more transition points of cache can be seen in tiled-Matmult of Figure 3.1(f). When the problem size is large enough, the only working set left is the tile size. In other words, the excess of the working set of a tile can be turned off to save power.

3.2.3 Relations of WCET, Useful Cache and Leakage Power

The compiler-based useful cache calculation algorithm in the PWCET framework can provide several different transition points of working sets. Choosing the largest working set can avoid affecting the WCET bound and gain power saving at the same time as shown in the second subsection. To study how other smaller working sets will
Figure 3.2: WCET penalty and Power Saving vs. Useful Cache Size
affect the WCET and also the relation with power savings, experiments are conducted from 0% cache turned off to all cache turned off by a step of 5% for each benchmark.

To evaluate the effect on WCET, we define the WCET penalty as following:

$$\text{Penalty}_{\text{WCET}} = \frac{\text{Bound}_P}{\text{Bound}_T}$$

where $\text{Bound}_P$ represents the WCET bound obtained by PWCET analysis based on a particular resized cache, $\text{Bound}_T$ is the WCET bound of traditional static timing analysis with original cache size. Hence, $\text{penalty} = 1$ means no effect at all, $\text{penalty} > 1$ means the WCET bound increases due to the cache reduction.

For each graph in Figure 3.2, grey histogram denotes $\text{penalty}$ and black histogram denotes percentage of power saving. Benchmark $\text{Cnt}$ only has spatial reuse, the penalty remains at 1 regardless of reduced cache size based on different working sets.

For other benchmarks, the $\text{penalty}$ increases along with the cache reduction and the change of penalty happened at transition points of working sets. For example, in $\text{Matmult}$, transition happened at 85% and 90% of cache with corresponding $\text{penalty}$ 1.10 and 1.54.

Note that the transition points with $\text{penalty} > 1$ can provide higher power efficiency. Thus, if the penalty can be accepted by the requirements of systems, then these smaller working sets can take place of the original one to provide higher power efficiency. For example, some soft real-time systems such as video streaming systems can tolerate some violations. However, hard real-time systems can not tolerate any violation.

For all benchmarks except $\text{Cnt}$, there is a sharp increase of WCET $\text{penalty}$ when 100% cache are turned off. Here we assumed that only one single cache line is left when 100% cache are turned off. There is a dramatically increase of worst case
### Table 3.2: PWCET for Multi-core Analysis

<table>
<thead>
<tr>
<th>Benchmarks</th>
<th>Traditional Static Timing Analysis / PWCET</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Cache</td>
</tr>
<tr>
<td>OMP_cnt</td>
<td>512/16</td>
</tr>
<tr>
<td>OMP_matmult</td>
<td>512/256</td>
</tr>
<tr>
<td>OMP_tiledMM</td>
<td>512/256</td>
</tr>
</tbody>
</table>

execution time because almost all cache references become misses due to a tiny cache size. On the contrary, the fraction of power savings steadily increases along with cache reduction.

#### 3.2.4 Validation of Multi-core Analysis

To validation the multi-core analysis, we have to model the execution of OpenMP programs in sequential code. Because we are limited by the analysis tool. We do not have a multi-core version of the WCET tool to analyze OpenMP programs.

The key point is to model the memory access pattern and cache utilization, so we adapt affine computation into a round-robin fashion to model the OpenMP execution. For example, suppose we have 1000 $i$ iterations in matrix multiplication, then we modify the code to make the $i$ loop execution order to 0, 500, 1, 501, ..., 499, 999. Then we perform WCET analysis on this OpenMP-like sequential code. The generated bound is not the *worst-case execution time*, but we argue that our multi-core analysis works if this bound does not change after resizing cache.

Table 3.2 shows the results of multi-core analysis, we only choose *Cnt*, *matmult* and *tiled-Matmult* since others are not feasible for OpenMP parallelization.
OMP\_cnt has no temporal reuse, so the useful cache size does not change. For other test cases, the useful cache size doubled because the reuse happened at the parallel loop.
Chapter 4: Conclusions

In this thesis, we propose a novel Power-aware WCET analysis framework based on a compiler analysis to determine the useful cache size for improving power efficiency. We extend the useful cache calculation algorithm to resizing useful cache in WCET analysis and propose the multi-core analysis of parallel programs. The compiler analysis based algorithm can be incorporate with both static timing analysis and measurement-based analysis. Experimental results show that the PWCET framework can obtain significant power saving without affecting WCET bounds. Besides, it can also provide different transition points of cache size in which power consumption can be further reduced but WCET bound may get increased to some extent. These different combinations of power saving and WCET bound are useful for energy focused systems.
Bibliography

[1] Cacti: Cache access time and power model.


