H/S Codesign: Performance Analysis of Software

© Copyright by Margarida Jacome

The University of Texas at Austin

Performance Analysis

Princeton: Sharad Malik, Wayne Wolf, Andrew Wolf (Cinderella Project)

1 Introduction
2 Microarchitectural Modeling
   ♦ Cache
   ♦ CPU Pipelining
3 Experimental Results
Worst Case Analysis

1. Worst case execution time (WCET) of a process running on a given processor
   - In real time systems: designer must verify if the WCET satisfies the timing deadlines
     » Many real-time OSs rely on this information for process scheduling
   - In embedded system design: the WCET is often required to decide on Hw/Sw partitioning

2. Determining the actual WCET of a program
   - Impractical: requires simulating all possible combinations of input data values and initial system states
   - Alternative: obtaining an estimate of the actual WCET by a static analysis of the program
     - Estimated WCET: must be tight and conservative (informative upper bound)
Problem Formulation

Determining the estimated WCET of a given program on a given hardware system, assuming uninterrupted execution

- Hardware system: includes the microprocessors and the memory systems

1. Problem has two components
   - Program path analysis
   - Microarchitectural modeling

Estimating WCET

1. The problem of finding a program’s estimated WCET is in general undecidable
2. Conditions for this problem to be decidable
   - Absence of recursive function calls
   - Bounded loops
Program Path Analysis

In order to tighten the estimated WCET, it is necessary to remove logically unfeasible program paths:
- path information provided by programmer
- scope of path information provided by designer may have direct impact on the tightness of the estimated WCET

Statically feasible paths and programmer annotations:
- loop bounds;
- interactions among different program statements (e.g., mutual exclusion of two statements)

Estimating WCET

Approach: the problem of solving the estimated WCET is converted into a set of Integer linear programming (ILP) problems.

Discussion assumes (for now) a simple (pessimistic) microarchitectural model:
- execution time of an instruction is constant
  - every instruction fetch results in a cache miss
A basic block is a maximum contiguous sequence of instructions, with a single entry point at its start and a single exit point at its end.

Program CFG

Estimating WCET: ILP Formulation

Assumption: each instruction takes a constant time to execute.

Total execution time = \sum_{i=1}^{N} c_i x_i

- N = number of BBs
- c_i = execution time of basic block B_i
- x_i = execution count of basic block B_i
**Estimating WCET: ILP Formulation**

Assumption: each instruction takes a constant time to execute.

Total execution time: \[ \sum_{i} c_i x_i \]

- \( N \) = number of BBs
- \( c_i \) = execution time of basic block \( B_i \)
- \( x_i \) = execution count of basic block \( B_i \)

**Constraints**

- Program structure
- Values of program variables

**Execution Count of BBs**

Two types of constraints:

1. **Structural constraints**
   - derived from the program’s control flow graph (CFG)
2. **Program functionality constraints**
   - provided by the user to specify loop bounds, and other path information
Structural Constraints

- Each node $B_i$ defines an execution count variable $x_i$.

- Each edge defines a variable that counts the number of times the control flow passes through that edge.

```
while (k < 10)

if (ok)
    j++; j=0;
    ok = true;

k++; x5
```

CFG

```
B1
s=k; x1

B2
while (k < 10)
    x3

B3
if (ok)
    j=0;
    ok = true;

B4
j++; x3

B5
k++; x5

B6
x6

B7
r=j; x7
```

Margarida Jacome - UT Austin - 19
**Structural Constraints**

Analysis of CEGSMilar to standard network flow problem.

Execution count of node B_i is equal to:

- Number of times the control enters the node (inflow)
- Number of times the control exits the node (outflow)

d_1 = 1 specifies that fragment is to be executed once

\[ x_1 = d = \overline{d} \]
Structural Constraints

\[ \begin{align*}
  d_1 &= 1 \\
  x_1 &= d \\
  x_2 &= d + d \\
  x_3 &= d + d \\
  x_4 &= d \\
  x_5 &= d \\
  x_6 &= d + d \\
  x_7 &= d
\end{align*} \]
Constraints Execution Count of BBs

1. **Structural constraints**
   - derived from the program’s control flow graph (CFG)

2. **Program functionality constraints**
   - provided by the user to specify loop bounds, and other path information

Program Functionality Constraints

Loop Bounds

Since \( k \) is positive before it enters the loop, the loop body will be executed between 0 and 10 times. The loop is:

\[
0 \leq x_3 \leq 10 \times x_1
\]

```c
/*k >= 0*/
s=k;
while (k < 10){
  if (ok)
    j++;
  else {
    j=0;
    ok = true;
  }
  k++;
}
r=j;
```
Program Functionality Constraints

Other path information

Ex.1: Else statement (B5) can be executed at most once inside the loop

\[ x_5 \leq 1 \times x_3 \]

/*k >= 0*/  
\[ s=k; \]
while (k < 10){  
  if (ok)
    j++;  
  else{
    j=0;  
    ok = true;  
  }
}

Ex.2: If the else statement (B5) is executed the loop will be executed exactly 5 times

\[ (x_5 = 0) \times x_5 \geq 0 \times 0 = 5 \times x_1 \]

Non linear constraint

disjunction of linear constraint sets
(at least one constraint must be satisfied)
Solving the Constraints

Because of the AND and OR operators, the program functionality constraints may, in general, be a disjunction of conjunctive constraint sets.

To solve the estimated WCET, each set of the functionality constraints is combined with the set of structural constraints.

- the combined set is passed to the ILP solver with \( (1) \) to be maximized

\[
\text{Total execution time} = \sum_{i=1}^{N} c_i x_i
\]

ILP solver returns max value of expression and the BB counts

The procedure is repeated for every functionality constraint set.

The maximum over all these running times is the estimated WCET.
**Time Complexity**

1. Total time required to solve the estimated WCET depends on the number of functionality constraint sets and the time to solve each constraint set...

2. Number of functionality constraint sets doubles for each functionality constraint with a disjunction operator.

---

**Microarchitectural Modeling**

Objective: model the CPU pipeline and the cache memory systems and determine the execution times ($c_i$) of the BBs.

Assumption: a cache miss stalls the entire CPU pipeline.

Allows to divide the execution of a BB into parts:

- Time for cache penalty (program cache)
- Memory-invariant ideal execution time in the CPU pipeline
**Directed-Mapped Instruction Cache Analysis**

1. Previous cost function (total execution time) is modified.
2. Linear constraints representing the cache memory behavior (cache constraints) are added.

---

**Modified Cost Function**

1. Without cache, the factors that affect total execution time are the execution counts of BBs and their execution times.
2. With cache, the execution time of an instruction may vary, depending on whether it results in a cache hit or a cache miss.
   - For each instruction, subdivide original execution counts into counts of the *numbers of cache hits and cache hits*.
   - Consider the cache hit and cache miss execution time of each instruction.
**Modified Cost Function**

1. **Instructions inside a BB**
   - have the same execution count
   - may have different cache miss and cache hit counts
     - they may map to different cache lines, each of which may have different cache activity

Different grouping of instructions is utilized for cache analysis as a new type of atomic structure

---

**Line-Block (l-block)**

1. Defines a contiguous sequence of instructions within the same BB that are mapped to the same line in the instruction cache
   - the instructions within an l-block will always be executed the same number of times and have the same cache hit and cache miss counts

Example:
- CFG with BBs
- Instruction cache with lines
**Identifying l-blocks**

1. For each basic block, find all the cache lines that instructions within it map to.
   - Add an entry on these cache lines in the cache table.

**Line-Block (l-block)**

1. Basic block $B_1$ is partitioned into three $l$-blocks: $B_{1,1}$, $B_{1,2}$ and $B_{1,3}$.
### Line-Blocks (l-blocks)

1. Two basic blocks are said to conflict with each other if the execution of one l-block will displace the other l-block in the instructions cache.
   - For this to happen, they must map to the same cache line.
   - Example: L-block $B_{1,1}$ conflicts with $B_{3,1}$.

```
<table>
<thead>
<tr>
<th>Cache line</th>
<th>Basic Block</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>$B_{1,1}$</td>
</tr>
<tr>
<td>1</td>
<td>$B_{1,2}$</td>
</tr>
<tr>
<td>2</td>
<td>$B_{1,3}$</td>
</tr>
<tr>
<td>3</td>
<td>$B_{2}$</td>
</tr>
</tbody>
</table>
```

### Line-Blocks (l-blocks)

1. L-Block $B_{2,2}$ does not conflict with any other blocks (once it is in cache, it will never be displaced).

```
<table>
<thead>
<tr>
<th>Cache line</th>
<th>Basic Block</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>$B_{1,1}$</td>
</tr>
<tr>
<td>1</td>
<td>$B_{1,2}$</td>
</tr>
<tr>
<td>2</td>
<td>$B_{1,3}$</td>
</tr>
<tr>
<td>3</td>
<td>$B_{2}$</td>
</tr>
</tbody>
</table>
```
**Line-Block (l-block)**

1. L-blocks B\(_{1.3}\) and B\(_{2.1}\) occupy only a partial cache line.
2. A cache miss during the execution of either l-block will cause the system to load the instructions of both l-blocks into cache. Later executions will be cache hits.

### Execution Count of L-Blocks

1. Execution count of l-block B\(_{i,j}\) is denoted as \(x_{i,j}\).
2. Cache hit count is denoted \(x_{i,j}^{hit}\).
3. Cache miss count is denoted \(x_{i,j}^{miss}\).

\[ x_i = x_{i,j} = x_{i,j}^{hit} + x_{i,j}^{miss}, \quad 1 \leq j \leq n_i \]
Total Execution Time

\[
\text{Total execution time} = \sum_i \sum_j \left( c_{i,j} \times x_{i,j}^{\text{hit}} + c_{i,j} \times x_{i,j}^{\text{miss}} \right)
\]

- \( N \) = number of BBs
- \( n_i \) = number of blocks of block \( B_i \)
- \( x_{i,j}^{\text{hit}} \) = cache hit count of block \( B_{i,j} \)
- \( x_{i,j}^{\text{miss}} \) = cache miss count of block \( B_{i,j} \)
- \( c_{i,j}^{\text{hit}} \) = hit cost of block \( B_{i,j} \)
- \( c_{i,j}^{\text{miss}} \) = miss cost of block \( B_{i,j} \)

(discussed later CPU pipeline)

Constraints

- No changes to previous structural and program functionality constraints

\[
\text{Total execution time} = \sum_i \sum_j \left( c_{i,j} \times x_{i,j}^{\text{hit}} + c_{i,j} \times x_{i,j}^{\text{miss}} \right)
\]

Structural
- \( d_1 = 1 \)
- \( x_1 = d = d + d \)
- \( x_2 = d + d = d + d \)
- \( x_3 = d + d = d + d \)

Functional
- \( (x_5 = 0) \times x_5 \geq 1 \) & \( x_5 = 5 \times x_5 \)
**Cache Constraints**

Additional “cache constraints” required...

For each cache line,

If there is only one block $B_k$ that maps to it, the first execution of the block may cause a cache miss, and all subsequent executions will result in cache hits.

Let $x_{k,l}^{\text{hit}}$ be the hit rate in cache line $k$ for basic block $B_l$, and $x_{k,l}^{\text{miss}}$ be the miss rate.

The total execution time is given by:

$$\sum_{i} \sum_{j} (c_{i,j}x_{i,j}^{\text{hit}} + c_{i,j}^{\text{miss}}x_{i,j}^{\text{miss}})$$

### CFG

<table>
<thead>
<tr>
<th>Cache line</th>
<th>Basic Block</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>$B_1$, $B_1$, $B_3$, $B_3.1$</td>
</tr>
<tr>
<td>1</td>
<td>$B_1$, $B_1$, $B_3$, $B_3.2$</td>
</tr>
<tr>
<td>2</td>
<td>$B_1$, $B_1$, $B_3$, $B_3.2$</td>
</tr>
<tr>
<td>3</td>
<td>$B_1$, $B_1$, $B_3$, $B_3.2$</td>
</tr>
</tbody>
</table>

### Cache line

<table>
<thead>
<tr>
<th>Cache line</th>
<th>Basic Block</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>$B_1$, $B_1$, $B_3$, $B_3.1$</td>
</tr>
<tr>
<td>1</td>
<td>$B_1$, $B_1$, $B_3$, $B_3.2$</td>
</tr>
<tr>
<td>2</td>
<td>$B_1$, $B_1$, $B_3$, $B_3.2$</td>
</tr>
<tr>
<td>3</td>
<td>$B_1$, $B_1$, $B_3$, $B_3.2$</td>
</tr>
</tbody>
</table>

Margarida Jacome - UT Austin - 1999
Cache Constraints

For each cache line:

- If two or more conflicting blocks (e.g., \( B_1 \) and \( B_2 \)) map to the execution of one of them, this will lead to an entire cache line miss.

\[ x_{1,3}^{\text{miss}} x_1^{\text{miss}} \leq 1 \]

### Cache line | Basic Block
---|---
0 | \( B_{1.1} B_1 \) \( B_1 B_{3.1} \)
1 | \( B_{1.2} B_1 \) \( B_1 B_{3.2} \)
2 | \( B_{1.3} B_1 \) \( B_2 B_{2.1} \)
3 | \( B_2 B_2 \)

Margarida Jacome - UT Austin - 19
Cache Conflict Graph

A cache conflict graph is constructed for every cache line that contains two or more conflicting l-blocks.

Nodes:
- a start node \( s \) and an end node \( e \) representing the start and the end of the program
- a node \( B_{k,l} \) for every l-block that maps to that cache line

Edges:
- A directed edge from node \( B_{k,l} \) to node \( B_{m,n} \) is added if there is a path in the CFG from BB \( B_k \) to BB \( B_m \) without passing through the BB of any other l-blocks of the same cache line (i.e., CCG).
Cache Conflict Graph

Edges

A directed edge from node $B_{k,l}$ to node $e$ is added if there is a path in the CFG from BB $B_k$ to the end of the CFG without passing through the BB of any other l-blocks of the same CCG.

Possible CCG for two conflicting l-blocks
Cache Conflict Graph

For each edge, a variable is assigned representing the number of times that the control flow passes through that edge. CCG has the structure of a network flow graph.

Two constraints are constructed for each node $B_{i,j}$:

$$x_i = \sum_{u,v} P_{(u,v,i,j)} = \sum_{u,v} P_{(i,j,u,v)}$$

The execution count of $B_{i,j}$ enters $P_{(u,v,i,j)}$ and exits $P_{(i,j,u,v)}$. 
**Cache Conflict Graph**

Constraints between the p-variables to the program structural and program functionality constraints (via variables).

Two constraints are constructed for each node \( B_{i,j} \):

\[
x_i = \sum_{u,v} p(u,v \rightarrow i,j) \sum_{j,v} p(i,j \rightarrow u,v)
\]

The p-variable \( p(k.l, k.l) \) represents the number of times that the control flows into block \( B_{k.l} \) after executing block \( B_{k.l} \) without entering any other conflict block between.

The constraint for execution count of \( B_{k.l} \):

\[
x_{k.l} = p(k.l, k.l) + p(s, k.l) - p(k.l, k.l) + p(s, k.l)
\]
**Cache Conflict Graph**

Starting condition indicating that the program is executed once

\[
\sum_{u,v} p(s,u,v) = 1
\]

**Cache Constraints**

Bounding the cache miss penalty

\[
x_i = \sum_{u,v} p(u,v,i) = \sum_{u,v} p(i,v)
\]

\[
\sum_{u,v} p(s,u,v) = 1
\]

\[
p_{(k,l,k,l)} + p_{(k,l,m,k)} x_{k,l} \geq x_{k,l} \quad \text{hit} = (l,k)
\]

\[
x_{k,l} \text{hit} = (l,k)
\]

[Diagram of Cache Conflict Graph]

[Diagram of Cache Constraints]
**Additional Constraints**

- **p-variables** represent whether or not there is a path between two conflicting blocks.

The path represented by the p-variable may actually pass through a sequence of basic blocks...

- Maximum value of the p-variable is bounded by the minimum of these BBs execution counts.

Or ILP may return unfeasible l-block count and overly pessimistic WCET.

---

**Example**

```
X_4 = p(s, e) + p(4.1, e) + p(4.1, 4.1) + p(4.1, 7.1) + p(7.1, 4.1) + p(7.1, 7.1) + p(7.1, e) + p(7.1, 4.1) + p(7.1, 7.1) + p(7.1, e)
X_7 = p(s, e) + p(7.1, e) + p(4.1, e) + p(4.1, 4.1) + p(4.1, 7.1) + p(7.1, 4.1) + p(7.1, 7.1) + p(7.1, e) + p(7.1, 4.1) + p(7.1, 7.1) + p(7.1, e)
```
**Example**

Cache Constraints

\[
X_4 = p(s_1) + p(d_1, 1) + p(7_1, 4) = p(4_1) + p(4_1, 4) + p(d_1, 1)
\]

\[
X_7 = p(s_1) + p(d_1, 4) + p(4_1, 7) = p(7_1) + p(7_1, 4) + p(7_1, 4)
\]

\[
p(s_1) + p(s_7, e) + p(e) = 1
\]

\[
X_4 = p(s_1) + p(4_1, 4_1) + p(7_1, 7_1) = p(s_1) + p(4_1, 4_1)
\]

**Functional Constraints:**
- Both loops will be executed 10 times each time they are entered.
- Basic block B4 will be executed 9 times each time the outer loop is entered.

\[
x_3 = 10x_1 \quad x_7 = 10x_5 \quad x_4 = 9x_1
\]
Example

WC Solution: maximum number of cache misses: impossible execution sequence

Example

Placing bounds on variables:
- All paths entering loop must pass through the preheader
- Sum of the flows at most equal to execution count of loop
**CPU Pipelining**

Assumption: time required to execute a sequence of instructions in the CPU pipeline is always a constant through the execution of the program.

\[
\text{Total execution time} = \sum_{i} \sum_{j} \left( c_{i,j}^{\text{hit}} x_{i,j}^{\text{hit}} + c_{i,j}^{\text{miss}} x_{i,j}^{\text{miss}} \right)
\]

- \(N\) = number of BBs
- \(n_i\) = number of blocks of block \(B_i\)
- \(x_{i,j}^{\text{hit}}\) = cache hit count of block \(B_{i,j}\)
- \(x_{i,j}^{\text{miss}}\) = cache miss count of block \(B_{i,j}\)
- \(c_{i,j}^{\text{hit}}\) = hit cost of block \(B_{i,j}\)
- \(c_{i,j}^{\text{miss}}\) = miss cost of block \(B_{i,j}\)

- Found by adding up the effective execution time of the instructions in the block.
- Some instructions (especially floating-point instructions) are data-dependent.
- Conservative approach: worst-case effective execution time of floating-point operation (worst-case execution time of floating-point operation may be 30-40% its average execution time).
- Additional time is added to the last block of each BB so that all buffered load/store instructions are completed before the control reaches the end of the BB.
**CPU Pipelining**

\[
\begin{align*}
\text{hit cost of } l\text{-block } B_{i,j} & = c_{i,j}^{\text{hit}} \\
\text{miss cost of } l\text{-block } B_{i,j} & = c_{i,j}^{\text{miss}}
\end{align*}
\]

- Equal to corresponding cost plus the time needed to load the instructions of the block(s) from cache memory.

---

**Cinderella**

1. Estimates the WCET of programs running on an Intel QT960 development board containing:
   - 20 MHz Intel i960
     - 32-bit RISC processor
     - on-chip 512-byte direct-mapped instruction cache, organized as 32x16-byte lines
     - 4 stage instruction pipeline
     - floating point unit
   - 128K of main memory
1. Uses a public domain ILP solver
Experimental Results

"Actual WCET" was identified by authors, and the program execution time was measured for this worst case data set.

Benchmarks

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>Lines</th>
<th>Bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>check_data</td>
<td>Example from Park’s Thesis</td>
<td>17</td>
<td>88</td>
</tr>
<tr>
<td>picksrt</td>
<td>Insertion</td>
<td>15</td>
<td>80</td>
</tr>
<tr>
<td>line</td>
<td>Line drawing routine from Park's Thesis</td>
<td>143</td>
<td>1,556</td>
</tr>
<tr>
<td>circle</td>
<td>Circle drawing routine from Park's Thesis</td>
<td>88</td>
<td>1,588</td>
</tr>
<tr>
<td>fft</td>
<td>Fast Fourier transform</td>
<td>56</td>
<td>544</td>
</tr>
<tr>
<td>des</td>
<td>Data Encryption Standard</td>
<td>185</td>
<td>1,796</td>
</tr>
<tr>
<td>fullsearch</td>
<td>MPEG2 encoder frame search routine</td>
<td>204</td>
<td>1,436</td>
</tr>
<tr>
<td>whetstone</td>
<td>Whetstone benchmark</td>
<td>245</td>
<td>2,760</td>
</tr>
<tr>
<td>drystone</td>
<td>Drystone benchmark</td>
<td>480</td>
<td>1,360</td>
</tr>
<tr>
<td>matgen</td>
<td>Matrix routines Linpack benchmark</td>
<td>50</td>
<td>248</td>
</tr>
</tbody>
</table>
## Experimental Results

### Measured WCET

<table>
<thead>
<tr>
<th>Function</th>
<th>Measured WCET</th>
<th>Estimated WCET with cache analysis</th>
<th>Estimated WCET without cache analysis</th>
</tr>
</thead>
<tbody>
<tr>
<td>check_data</td>
<td>$4.41 \times 10^2$</td>
<td>$4.91 \times 10^2$</td>
<td>$11.9 \times 10^2$</td>
</tr>
<tr>
<td>picksrt</td>
<td>$1.79 \times 10^3$</td>
<td>$1.82 \times 10^3$</td>
<td>$5.01 \times 10^3$</td>
</tr>
<tr>
<td>line</td>
<td>$4.85 \times 10^3$</td>
<td>$6.09 \times 10^3$</td>
<td>$9.15 \times 10^3$</td>
</tr>
<tr>
<td>circle</td>
<td>$1.45 \times 10^4$</td>
<td>$1.53 \times 10^4$</td>
<td>$1.59 \times 10^4$</td>
</tr>
<tr>
<td>fft</td>
<td>$2.08 \times 10^4$</td>
<td>$2.71 \times 10^4$</td>
<td>$4.04 \times 10^4$</td>
</tr>
<tr>
<td>des</td>
<td>$2.42 \times 10^4$</td>
<td>$3.66 \times 10^4$</td>
<td>$6.69 \times 10^4$</td>
</tr>
<tr>
<td>fullsearch</td>
<td>$6.25 \times 10^4$</td>
<td>$9.57 \times 10^4$</td>
<td>$29.0 \times 10^4$</td>
</tr>
<tr>
<td>whetstone</td>
<td>$6.83 \times 10^4$</td>
<td>$10.2 \times 10^4$</td>
<td>$14.9 \times 10^4$</td>
</tr>
<tr>
<td>drystone</td>
<td>$5.52 \times 10^5$</td>
<td>$7.53 \times 10^5$</td>
<td>$13.3 \times 10^5$</td>
</tr>
<tr>
<td>matgen</td>
<td>$9.28 \times 10^5$</td>
<td>$10.9 \times 10^5$</td>
<td>$17.2 \times 10^5$</td>
</tr>
</tbody>
</table>

## Experimental Results

### Estimated WCET with cache analysis

<table>
<thead>
<tr>
<th>Function</th>
<th>Estimated WCET with cache analysis</th>
</tr>
</thead>
<tbody>
<tr>
<td>check_data</td>
<td>$4.91 \times 10^2$</td>
</tr>
<tr>
<td>picksrt</td>
<td>$1.82 \times 10^3$</td>
</tr>
<tr>
<td>line</td>
<td>$6.09 \times 10^3$</td>
</tr>
<tr>
<td>circle</td>
<td>$1.53 \times 10^4$</td>
</tr>
<tr>
<td>fft</td>
<td>$2.71 \times 10^4$</td>
</tr>
<tr>
<td>des</td>
<td>$3.66 \times 10^4$</td>
</tr>
<tr>
<td>fullsearch</td>
<td>$9.57 \times 10^4$</td>
</tr>
<tr>
<td>whetstone</td>
<td>$10.2 \times 10^4$</td>
</tr>
<tr>
<td>drystone</td>
<td>$7.53 \times 10^5$</td>
</tr>
<tr>
<td>matgen</td>
<td>$10.9 \times 10^5$</td>
</tr>
</tbody>
</table>

### Estimated WCET without cache analysis

<table>
<thead>
<tr>
<th>Function</th>
<th>Estimated WCET without cache analysis</th>
</tr>
</thead>
<tbody>
<tr>
<td>check_data</td>
<td>$11.9 \times 10^2$</td>
</tr>
<tr>
<td>picksrt</td>
<td>$5.01 \times 10^3$</td>
</tr>
<tr>
<td>line</td>
<td>$9.15 \times 10^3$</td>
</tr>
<tr>
<td>circle</td>
<td>$1.59 \times 10^4$</td>
</tr>
<tr>
<td>fft</td>
<td>$4.04 \times 10^4$</td>
</tr>
<tr>
<td>des</td>
<td>$6.69 \times 10^4$</td>
</tr>
<tr>
<td>fullsearch</td>
<td>$29.0 \times 10^4$</td>
</tr>
<tr>
<td>whetstone</td>
<td>$14.9 \times 10^4$</td>
</tr>
<tr>
<td>drystone</td>
<td>$13.3 \times 10^5$</td>
</tr>
<tr>
<td>matgen</td>
<td>$17.2 \times 10^5$</td>
</tr>
</tbody>
</table>

Tighter bounds

Small integer program