Static energy consumption analysis for LLVM IR programs

Neville Grech, Kyriakos Georgiou, James Pallister, Steve Kerrison, Jeremy Morse and Kerstin Eder

University of Bristol

February 15, 2016

Motivation

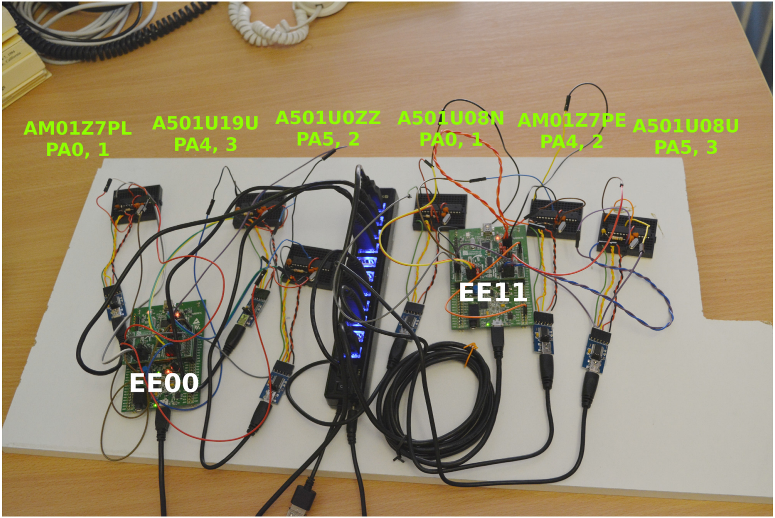
Software has final control of energy consumption:

- Instructions executed on processor causes circuit switching.
- Operation of software determines what clock domains can be gated.
- Amount of work performed by software determines how long the system draws power for.

Energy transparency enables decision making:

- Making energy information available to the developer allows for energy-based design decisions.
- Performing measurements requires dual-skill developers, instrumentation, and actual working hardware, presenting a burden to development.
- A solution then, is energy estimation from software.

Measuring energy is time consuming and complex.



Summary of our technique.

- Produce basic-block energy costs using processor energy model, and compiler infrastructure.
- Simplify program to make it ameanable to analysis.
- Extract cost relations (CRs), characterising the energy consumed by the program.
- Solve CRs and infer a closed form formula.

Summary of our technique.

- Produce basic-block energy costs using processor energy model, and compiler infrastructure.
- Simplify program to make it ameanable to analysis.
- Extract cost relations (CRs), characterising the energy consumed by the program.
- Solve CRs and infer a closed form formula.

Example, Levenshtein distance algorithm, i.e.:

- Initialize distance-table for string, length A.
- Initialize distance-table for string, length B.
- For each element of A, for each element of B, calculate insert/delete/modify distance.

Summary of our technique.

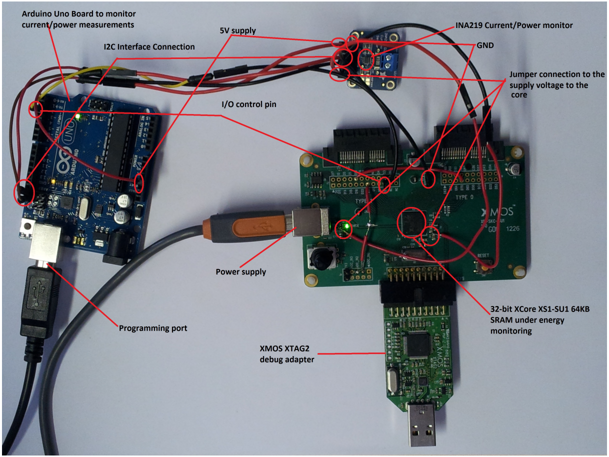
- Produce basic-block energy costs using processor energy model, and compiler infrastructure.
- Simplify program to make it ameanable to analysis.
- Extract cost relations (CRs), characterising the energy consumed by the program.
- Solve CRs and infer a closed form formula.

Example, Levenshtein distance algorithm, i.e.:

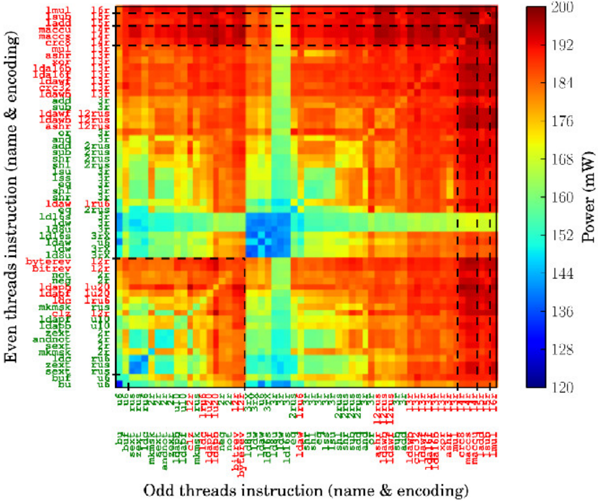
- Initialize distance-table for string, length A.
- Initialize distance-table for string, length B.
- For each element of A, for each element of B, calculate insert/delete/modify distance.

$$(A(53B + 16) + 35B + 31)\ \mathrm{nJ}, \tag{1}$$

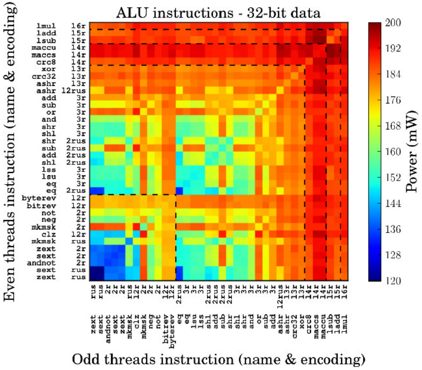
Energy measuring, set up.

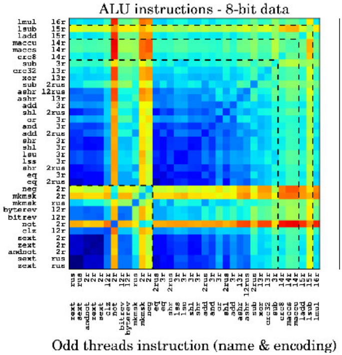


ISA Characterization



ISA Characterization





Understanding our processors

XMOS:

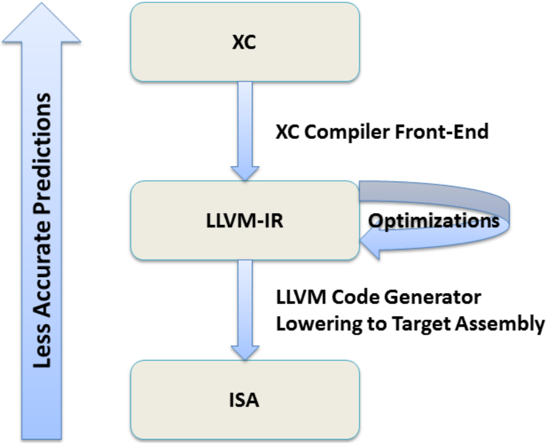
- Hardware multithreading microcontroller.
- No caches, branch prediction, or other timing unpredictable features.
- Designed for high frequency IO control.

ARM Cortex-M3:

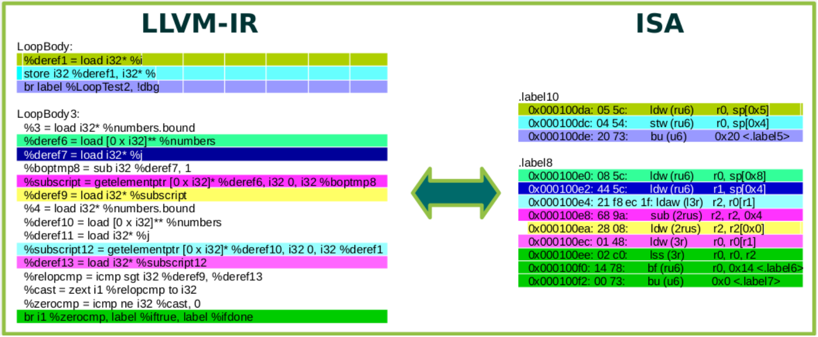
- Instructions grouped by characteristics (memory access, control flow, division, other).
- Average energy collected for each group of instructions.
- Energy assigned to LLVM-IR instructions.

This energy model is less accurate than the XCore model, but required less effort to produce.

Mapping LLVM IR blocks to machine instructions



Mapping LLVM IR blocks to machine instructions



Resource Analysis Example

Original program

```
int fact(int i) {
 if (i<=0)^a
   return 1^b;
 return (i*^d fact(i-1))^c;
}
```

Resource Analysis Example

Original program

```
int fact(int i) {
 if (i<=0)^a
   return 1^b;
 return (i*^d fact(i-1))^c;
}
```

Extracted cost relations

$$C_{fact}(i) = C_a + C_b \qquad if\ i <= 0$$

$$C_{fact}(i) = C_a + C_c(i) \qquad if\ i > 0$$

$$C_c(i) = C_d + C_{fact}(i-1)$$

Resource Analysis Example

Original program

```
int fact(int i) {
 if (i<=0)^a
   return 1^b;
 return (i*^d fact(i-1))^c;
}
```

Extracted cost relations

$$C_{fact}(i) = C_a + C_b \qquad if\ i <= 0$$

$$C_{fact}(i) = C_a + C_c(i) \qquad if\ i > 0$$

$$C_c(i) = C_d + C_{fact}(i-1)$$

- Substitute C_a, C_b, C_d with actual energy required to execute low level instructions.
- Solve relations using off the shelf solvers to obtain *closed form* solution.
- Result: $C_{fact}(i) = 4563 + 7878i\ pJ$.

Linguistic Abstraction

Various language subsets of LLVM IR have been formalised. For analysis purposes, we abstract LLVM IR as follows:

$$
\begin{array}{rlr}
\mathit{inst} = & \text{br } p \ BB_1 \ BB_2 & \text{conditional branch instruction} \\
\mid & x = \text{op } a_1..a_n & \text{generic side effect-free operation} \\
\mid & x = \phi \ \langle BB_1, x_1 \rangle .. \langle BB_n, x_n \rangle & \text{phi nodes} \\
\mid & x = \text{call } f \ a_1 \ .. \ a_n & \\
\mid & x = \text{memload} & \text{dynamic memory load operation} \\
\mid & \text{memstore} & \text{dynamic memory store operation} \\
\mid & \text{ret } a &
\end{array}
$$

where metavariable names p,f,a,x are predicates, function names, generic arguments and variables respectively.

Linguistic Abstraction

Various language subsets of LLVM IR have been formalised. For analysis purposes, we abstract LLVM IR as follows:

$inst = \text{br } p \; BB_1 \; BB_2$	conditional branch instruction
$\mid x = \text{op } a_1..a_n$	generic side effect-free operation
$\mid x = \phi \; \langle BB_1, x_1\rangle..\langle BB_n, x_n\rangle$	phi nodes
$\mid x = \text{call } f \; a_1 \; .. \; a_n$	
$\mid x = \text{memload}$	dynamic memory load operation
$\mid \text{memstore}$	dynamic memory store operation
$\mid \text{ret } a$	

where metavariable names *p,f,a,x* are predicates, function names, generic arguments and variables respectively.

Abstract semantics are standard, except:

- $x =$ memload simply invalidates x i.e., $x =?$.
- memstore may set any area in memory.

Building Cost Relations

Cost relations characterise the cost of executing a program or basic block, for example block a:

$$C_a(x, y) = S_a + C_b(x + n_1, y + m_1) + C_c(x + n_2, y + m_2) + \dots \text{ if condition}$$

Where

- C_a is the cost relation for block a,

Building Cost Relations

Cost relations characterise the cost of executing a program or basic block, for example block a:

$$C_a(x, y) = S_a + C_b(x + n_1, y + m_1) + C_c(x + n_2, y + m_2) + \ldots \text{ if condition}$$

Where

- C_a is the cost relation for block a,
- x and y input arguments to a (local variables or registers),

Building Cost Relations

Cost relations characterise the cost of executing a program or basic block, for example block a:

$$C_a(x, y) = S_a + C_b(x + n_1, y + m_1) + C_c(x + n_2, y + m_2) + \ldots \text{ if condition}$$

Where

- C_a is the cost relation for block a,
- x and y input arguments to a (local variables or registers),
- S_a is the static energy cost of executing the block, which we get using the mapper.

Building Cost Relations

Cost relations characterise the cost of executing a program or basic block, for example block a:

$$C_a(x, y) = S_a + C_b(x + n_1, y + m_1) + C_c(x + n_2, y + m_2) + \ldots \text{ if condition}$$

Where

- C_a is the cost relation for block a,
- x and y input arguments to a (local variables or registers),
- S_a is the static energy cost of executing the block, which we get using the mapper.
- C_b, C_c, etc. characterise the cost of executing *continuations* from block a, e.g., blocks that branch from a.

Building Cost Relations

Cost relations characterise the cost of executing a program or basic block, for example block a:

$$C_a(x, y) = S_a + C_b(x + n_1, y + m_1) + C_c(x + n_2, y + m_2) + \ldots \text{ if condition}$$

Where

- C_a is the cost relation for block a,
- x and y input arguments to a (local variables or registers),
- S_a is the static energy cost of executing the block, which we get using the mapper.
- C_b, C_c, etc. characterise the cost of executing *continuations* from block a, e.g., blocks that branch from a.
- Side condition reflects a branching predicate in the original program.

Building Cost Relations

Cost relations characterise the cost of executing a program or basic block, for example block a:

$$C_a(x, y) = S_a + C_b(x + n_1, y + m_1) + C_c(x + n_2, y + m_2) + \ldots \text{ if condition}$$

Where

- C_a is the cost relation for block a,
- x and y input arguments to a (local variables or registers),
- S_a is the static energy cost of executing the block, which we get using the mapper.
- C_b, C_c, etc. characterise the cost of executing *continuations* from block a, e.g., blocks that branch from a.
- Side condition reflects a branching predicate in the original program.

Next: How do we infer cost relations from existing programs?

Symbolic Evaluation

At the core of our resource analysis mechanism is a symbolic evaluation function *seval*.

- Given a block of code BB, and a variable x, $seval(BB, x)$ symbolically executes a slice from this block to produce a result.
- Abstracts away the effect of dynamic memory reads and writes, i.e., memload and memstore
- Hence we can produce simple expressions, which can be handled by existing solvers.

Symbolic Evaluation cont.

Symbolic evaluation is used:

- On branch predicates to infer side conditions for cost relations.
- To summarise the effect of executing a block on a variable (e.g. induction variable).

Example:

```
LoopIncrement:
  %postinc = add i32 %i.0, 1
  %exitcond = icmp eq i32 %postinc, %1
  br i1 %exitcond, label %return, label %LoopBody
```

In this case *seval*(BB,`%exitcond`) is (`%i.0`+1) =`%1`, found by traversing the structure of the LLVM block backwards.

Cost Relations Arguments

To simplify the analysis, we only consider variables and arguments that affect the control flow, e.g.:

```
int accumulate(int n) {
    int res = n;
    while (n--) res+=n;
    return res;
}
```

- Only the value of `n` affects the control flow, not `res`. CRs will therefore be parametric with `n`.
- Computed using data flow analysis techniques.

Analysing Nested Loop Structures

Certain classes of programs require further analysis/transformation.

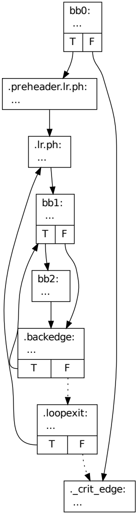
```
void bubbleSort(int numbers[], int array_size) {
  int i, j, temp;
  for (i = (array_size - 1); i >= 0; i--) {
    for (j = i; j > 0; j--)  {
      if (numbers[j-1] > numbers[j]) {
        temp = numbers[j-1];
        numbers[j-1] = numbers[j];
        numbers[j] = temp;
      }
    }
  }
}
```

Unfortunately, the high level loop structure is mangled by compiler optimisations.

LLVM IR (clang -O3)

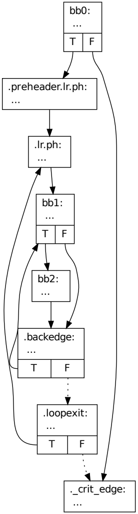
```
define void @bubbleSort(i32* nocapture %numbers, i32 %array_size)  {
bb0:
  %i.02 = add nsw i32 %array_size, -1
  %1 = icmp sgt i32 %i.02, 0
  br i1 %1, label %.preheader.lr.ph, label %._crit_edge
.preheader.lr.ph:
  %2 = sext i32 %i.02 to i64
  br label %.lr.ph
.loopexit:
  %i.0 = add nsw i32 %i.03, -1
  %3 = icmp sgt i32 %i.0, 0
  %indvars.iv.next = add i64 %indvars.iv, -1
  br i1 %3, label %.lr.ph, label %._crit_edge
.lr.ph:
  %indvars.iv = phi i64 [ %2, %.preheader.lr.ph ], [ %indvars.iv.next, %.loopexit ]
  %i.03 = phi i32 [ %i.02, %.preheader.lr.ph ], [ %i.0, %.loopexit ]
  br label %bb1
bb1:
  %indvars.iv4 = phi i64 [ %indvars.iv, %.lr.ph ], [ %indvars.iv.next5, %.backedge ]
  %j.01 = phi i32 [ %i.03, %.lr.ph ], [ %5, %.backedge ]
  %5 = add nsw i32 %j.01, -1
  %6 = sext i32 %5 to i64
  %7 = getelementptr inbounds i32* %numbers, i64 %6
  %8 = load i32* %7, align 4,
  %9 = getelementptr inbounds i32* %numbers, i64 %indvars.iv4
  %10 = load i32* %9, align 4,
  %11 = icmp sgt i32 %8, %10
  br i1 %11, label %bb2, label %.backedge
.backedge:
  %12 = icmp sgt i32 %5, 0
  %indvars.iv.next5 = add i64 %indvars.iv4, -1
  br i1 %12, label %bb1, label %.loopexit
bb2:
  store i32 %10, i32* %7, align 4, !tbaa !0
  store i32 %8, i32* %9, align 4, !tbaa !0
  br label %.backedge
._crit_edge:
  ret void
}
```

Extracting Loop Structure

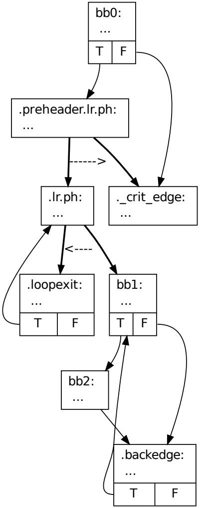


- The compiler turns nested loops into CFG structures without cover points (left).

Extracting Loop Structure



- The compiler turns nested loops into CFG structures without cover points (left).
- However, these can be transformed in a lot of cases (right).
- *Energy conservation semantics* preserved for typical `while` or `for` loops compiled by `clang` or `xcc`.



Benchmarks.

- Selection of benchmarks from existing suites .
  - Levenshtein distance, Matrix multiply (BEEBS).
  - Multiply-accumulate, JPEG DCT (WCET).
  - Additionally, insertion sort and base64 encoding algorithms.
- Each benchmark analyzed by our technique to produce a cost formula.
- Evaluated over a range of different input sizes to determine accuracy.

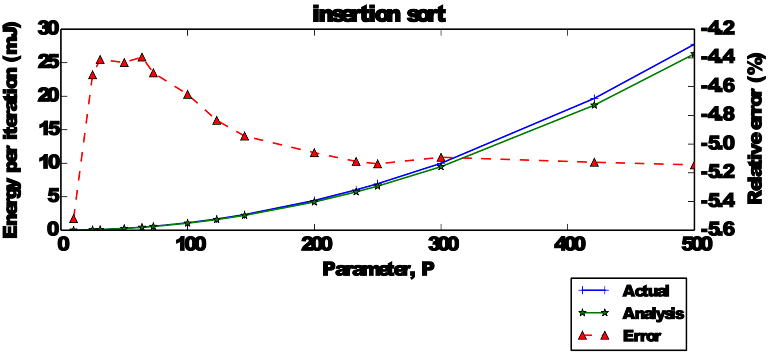
Formulae produced from benchmarks.

	ARM (nJ)	XMOS (nJ)
base64	$158 + 94 \cdot \left\lfloor \frac{P-1}{3} \right\rfloor$	$1270 + 734 \cdot \left\lfloor \frac{P-1}{3} \right\rfloor$
mac	$23P + 14$	$133P + 192$
levenshtein	$47AB + 14A + 31B + 44$	$559AB + 78A + 571 + \max(225B, 180B + 213)$
insertion sort	$25P^2 + 11P + 7.1$	$105P^2 + 30P + 75$
matrix multiply	$20P^3 + 13P^2 + 97P + 84$	$144P^3 + 200P^2 + 77P + 332$
jpegdct	54 mJ‡	463 mJ‡

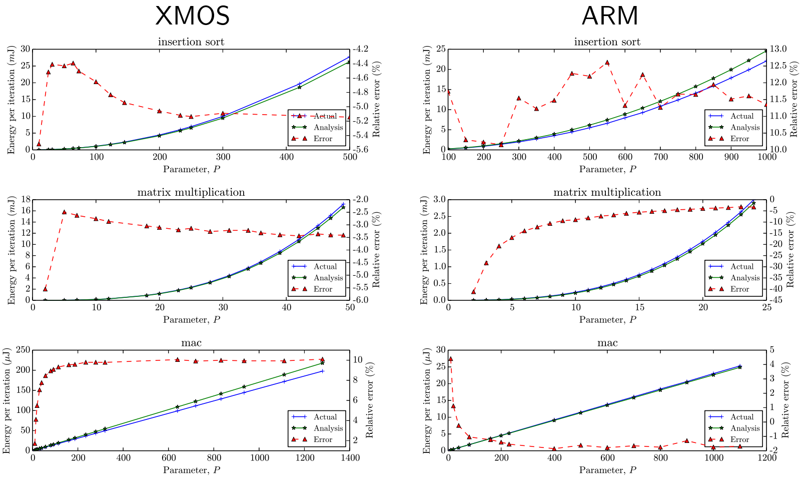
‡ This benchmark was not parametric, thus is not parameterised.

Table: Formulae and error values for each benchmark.

Example result.



Other results.



Discussion.

- Our energy estimations are within roughly 10% of measured energy for XMOS.
- Worse estimation for ARM, where we do not have a detailed energy model, but within 30% (most 15%).
- Generic technique (LLVM), requires either a detailed energy model, or as we have shown an approximation is sufficient too.
- The analysis itself requires little computational resources.
- Can be integrated into a development workflow.