Dynamic Compilation

Michael O’Boyle

March, 2014
Overview

• Dynamic Compilation

• Specialisation
  – DyC
  – Calpa

• Dynamic Binary translation
  – Dynamo
Dynamic techniques

• These techniques focus on delaying some or all of the optimisations to runtime

• This has the benefit of knowing the exact runtime control-flow, hotspots, data values, memory locations and hence complete program knowledge

• It thus largely eliminates many of the undecidable issues of compile-time optimisation by delaying until runtime

• However, the cost of analysis/optimisation is now crucial as it forms a runtime overhead. All techniques characterised by trying to exploit runtime knowledge with minimal cost
Background

- Delaying compiler operations until runtime has been used for many years

- Interpreters translates and execute at runtime

- Languages developed in the 60s eg Algol 68 allowed dynamic memory allocation relying on language specific runtime system to manage memory

- Lisp more fundamentally has runtime type checking of objects

- Smalltalk in the 80s deferred compilation to runtime to reduce the amount of compilation otherwise required in the 00 setting

- Java uses dynamic class loading to allow easy upgrading of software
Runtime specialisation

• For many, runtime optimisation is “adaptive optimisation”

• Although wide range of techniques, all are based around runtime specialisation. Constant propagation is a simple example.

• Specialisation is a technique that has been used in compiler technology for many years especially in more theoretical work

• Specialising an interpreter with respect to a program gives a compiler

• Can we specialise at runtime to gain benefit with minimal overhead? Statically inserted selection code vs parametrised code vs runtime generation.
Static code selection, parametrised and code generation

IF (N<M) THEN
DO I =1,N
    DO J =1,M
        ...
    ENDDO
ENDDO
ELSE
    DO J =1,M
        DO I =1,N
            ...
        ENDDO
ENDDO
ENDIF

IF (N<M) THEN
    U1 = N
    U2 = M
ELSE
    U1 = M
    U2 = N
ENDIF

doi =1,U1
    DO i2= 1,U2
        ...
    ENDDO
ENDDO

M. O’Boyle
Iterative Compilation
March, 2014
DyC

• One of the best known dynamic program specialisation techniques based on dynamic code generation.

• The user annotates the program defining where there may be opportunities for runtime specialisation. Marks variables and memory locations that are static within a particular scope.

• The system generates code that checks the annotated values at runtime and regenerates code on the fly.

• By using annotation, the system avoids over-checking and hence runtime overhead. This is at the cost of additional user overhead.
DyC

- Focuses on static runtime values and memory locations. Static binding-time analysis to identify static runtime expressions.

- Polyvariant division and specialisation to replicate control-flow path and divide into static and run-time variable sub-graphs.

- A specialised compiler is inserted at dynamic sub-graphs. At runtime it compiles code with values and stores this in a cache for later use and then executes it.

- Later invocations look up the code and use it. Uses full unrolling, strength reduction, zero/copy propagation, dead-assignment elimination and static loads.
Binding analysis examines all uses of static variables within scope.

Dynamic compiler exploits invariancy and specialises the code when invoked.
DyC Example

```c
make_static(CMTRIX,CROWS,CCOLS,CROW,CCOL); -- USER annotation
CROWS02 = CROWS/2; CCOLS02= CCOLS/2;
for(irow=0; irow<irows; ++irow){
    rowbase = irow-CROWS02;
    for(icol= 0; icol<icols;++icol){
        colbase = icol-CCOLS02; sum =0.0;
        for (CROW=0; CROW<CROWS; ++CROW){ -- unroll loop
            for (CCOL=0; CCOL <CCOLS; +CCOL){ -- unroll loop
                weight = cmatrix @[CROW]@[CCOL]; -- constant load
                x = image[rowbase+CROW][colbase+CCOL];
                weighted_x = x * weight;
                sum = sum + weighted_x;
                outbuf[irow][icol] = sum;
            }
        }
    }
}
```

M. O'Boyle Iterative Compilation March, 2014
DyC Example

Matrix cmatrix has the following structure:

\[
\begin{array}{ccc}
0 & 1 & 0 \\
1 & 0 & 1 \\
0 & 1 & 0 \\
\end{array}
\]

- When CROW and CCOL loop is unrolled, the static loads from cmatrix will be inlined
- Allows elimination of redundant computation
DyC Example

-- crow and ccol unrolled
-- weight accesses static load, is propagated and eliminated
x = image[rowbase][colbase];       -- crow = 0, ccol = 0
weighted_x = x * 0.0;               -- weight == cmatrix[0][0] == 0
sum = sum + weighted_x;

-- weight accesses static load, is propagated and eliminated
x = image[rowbase][colbase+1];     -- crow = 0, ccol = 1
weighted_x = x * 1.0;              -- weight == cmatrix[0][0] == 1
sum = sum + weighted_x;
...

M. O’Boyle
Iterative Compilation
March, 2014
DyC Example

for(irow=0; irow<irows; ++irow){
    rowbase = irow - 1;
    for(icol= 0; icol<icols2;++icol){
        colbase = icol - 1;

        x = image[rowbase][colbase+1]; -- It 1: crow=0, ccol=1
        sum = x;

        x = image[rowbase+1][colbase]; -- It 3: crow=1, ccol=0
        sum = sum + x;
        ...
        outbuf[irow][icol] = sum;
    }
}

M. O’Boyle  Iterative Compilation  March, 2014
DyC Example

- Asymptotic speedup and a range programs varies from 1.05 to 4.6

- Strongly depends on percentage of time spent in the dynamically compiled region. Varies from 9.9 to 100 %

- Low overhead from 13 cycles to 823 cycles per instruction generated

- Break-even point low.

- However relies on user intervention which may not be realistic in large applications

- Relies on user *correctly* annotating the code

M. O'Boyle

Iterative Compilation

March, 2014
Calpa for DyC

- Calpa is a system aimed at automatically identifying opportunities for specialisation without user intervention.

- It analyses the program for potential opportunities and determines the possible cost vs the potential benefit.

- For example, if a variable is multiplied by another variable which is known to be constant in a particular scope, then if this is equal to 0 or 1 then cheaper code may be generated.

- If this is inside a deep loop then a quick test for 0 or 1 outside the loop will be profitable.
Calpa for DyC

Calpa is a frontend to DyC.

It uses instrumentation to guide annotation insertion.
Calpa for DyC

i=0
L1: if i >= size goto L2
    uelem = u[i]
    velem = v[i]
    t = uelem * velem
    sum = sum + t
    i = i + 1
    goto L1
L2:

• At each statement calculate candidate static variables.

• Over larger region determine candidate division - all possible sub sets. Search for the best set based on value profile data using gradient search.
Calpa for DyC

- Instruments code and sees have often variables change value. Given this data determined the cost and benefit for a region of code

- Number of different variants, cost of generating code, cache lookup. Main benefit determined by estimating new critical path

- Explores all specialisation up to a threshold. Widely different overheads 2 seconds to 8 hours. Finds user choices and in two cases improves - from 6.6 to 22.6

- Calpa and DyC utilise selective dynamic code generation. Next look at fully dynamic schemes.
Dynamic Binary Translation

- The key idea is to take one ISA binary and translate it into another ISA binary at runtime.

- In fact this happens inside Intel processors where x86 is unpacked and translated into an internal RISC opcode which is then scheduled. The TransMeta Crusoe processor does the same. Same with IBM legacy ISAs.

- Why don’t we do this statically? Many reasons!

- The source ISA is legacy but the processor internal ISA changes. It is impossible to determine statically what is the program. It is not legal to store a translation. It can be applied to a local ISA for long term optimisation.
DYNAMO

• Focuses on binary to binary optimisations on the same ISA. One of the claims is that it allows compilation with -01 but overtime provides -03 performance.

• Catches dynamic cross module optimisation opportunities missed by a static compiler. Code layout optimisation allowing improved scheduling due to bigger segments. Branch alignment and partial procedural inlining form part of the optimisations.

• Aimed as way of improving performance from a shipped binary overtime.

• Has to use existing hardware - no additional fragment cache available

M. O’Boyle
Iterative Compilation
March, 2014
DYNAMO

- Initially interprets code. This is very fast as the code is native. When a branch is encountered check if already translated.

- If it has been translated jump and context switch to the fragment cache code and execute. Otherwise if hot translate and put in cache.

- Over time the working set forms in the cache and Dynamo overhead reduces -less than 1.5% on specint.

- Cheap profiling, predictability and few counters are necessary.

- Linear code structure in cache makes optimisation cheap. Standard redundancy elimination applied.
DYNAMO

Instruction

Interpret until taken branch

Look up branch target in cache

Start of trace condition?

No

Yes

Increment counter associated with branch target address

Counter value exceeds hot threshold?

No

Yes

Interpret + codegen until taken branch

End of trace condition?

No

Yes

Create new fragment and optimize it

Create new fragment and optimize it

Context switch

End of trace condition?

Emmit into cache, link with other fragments & recycle the associated counter

Jump to top of fragment in cache

Hit

End of trace condition?
DYNAMO

1. Interpret until taken branch.
2. Lookup branch target in cache.
   - If hit, jump to top of fragment in cache.
   - If not hit, start of trace condition?
     - Yes, increment counter associated with branch target address.
     - No, counter value exceeds hot threshold?
       - Yes, interpret + codegen until taken branch.
       - No, end of trace condition?
3. If not hit, context switch to Fragment Cache.
4. Emit into cache, link with other fragments & recycle the associated counter if end of trace condition.
5. Create new fragment and optimize it.
Control-flow in binary

Cross-module flow not picked up by compiler

DYNAMO

A

B

C

D

call

E

G

H

J

I

return
Layout of code in memory

The two functions may be very far apart in memory
Layout of trace in Dynamo’s fragment cache

DYNAMO

fragment body

trap to Dynamo

to B

to I

exit stubs
When there is a change of control-flow it is important to avoid returning to interpreter if at all possible.
Achieved by fragment linking

Also provides an opportunity for eliminating redundant compensation code

Essential optimisation factor 40!
DYNAMO percentage improvement

<table>
<thead>
<tr>
<th></th>
<th>trace selection</th>
<th>optimisation</th>
</tr>
</thead>
<tbody>
<tr>
<td>compress</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>go</td>
<td>-2</td>
<td>0</td>
</tr>
<tr>
<td>ijpeg</td>
<td>-2</td>
<td>0</td>
</tr>
<tr>
<td>li</td>
<td>16</td>
<td>6</td>
</tr>
<tr>
<td>mk88</td>
<td>18</td>
<td>5</td>
</tr>
<tr>
<td>perl</td>
<td>12</td>
<td>16</td>
</tr>
<tr>
<td>vortex</td>
<td>-1</td>
<td>0</td>
</tr>
<tr>
<td>def</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>avg</td>
<td>6</td>
<td>3</td>
</tr>
</tbody>
</table>

- Flush cache at phase change. Bail out if no steady control-flow. Optimising at a very low level. Complementary to DyC
Summary and conclusions

• All schemes allow specialisation at runtime to program and data

• Staged schemes such as DyC are more powerful as they only incur runtime overhead for specialisation regions

• JIT and DBT delay everything to runtime leaving little optimisation opportunities

• DyC focused on reducing ops - no investigation of processor behaviour

• Dynamo trace approach basis for JIT compilers