SPACE OPTIMIZATION FOR EMBEDDED PROCESSORS

Neil Johnson email: nej22@cl.cam.ac.uk
University of Cambridge Computer Laboratory, Cambridge, UK

THE NEED FOR COMPACT CODE

- **There is a massive growth in Mobile Computing**, with mobile phones now running Java VMs. As programs grow, download times become increasingly dominant.
  Bandwidth explosion!
- **Code size has a direct material cost** -> the smaller the program the smaller, and cheaper, the ROM needed for storage. Even on-chip deeply-embedded ROM space is a premium, where a significant portion of the chip is dedicated to the code ROM.
- **Speed improvement** if loop code bodies can fit into instruction cache.
- **Code size still a business factor** when comparing embedded processor products:
  Compact code -> more design wins!

PATTERNS WITHIN CODE

- **Procedural Abstraction** applied to Intermediate Code level:
  - Not constrained by register allocation or instruction scheduling
  - Higher level than previous attempts (concentrated on machine instruction level)
  - Lower level than source code, *normalising* (removing some variations due to source code layout, programming styles, etc.)

PATTERN PROCESSING

- **Identify** matching code patterns (see example)
  - Insensitive to register colouring and instruction scheduling
- **Match and Abstract:**
  - Use **VSDG** to match both control and dataflow patterns
  - Maintain a table of patterns, adding new entries for each new pattern discovered.
  - Ignore unsuitable patterns:
    - Too small (will be handled by code generation)
    - Too few repeats (cost of abstraction)
  - Generate compiler-generated abstract function F^A
  - Replace patterns with call to F^A, fixing up control and data dependencies.

```
...
   call [r10](r3,r6) -> r9
   sub r1, r2 -> r13
   add r1, r2 -> r15
   mul r1, r2 -> r17
   add r15, r13 -> r20
   div r20, r17 -> r19
...
...
   mul r1, r2 -> r7
   add r3, r5 -> r9
   sub r3, r5 -> r11
   mul r3, r5 -> r13
   add r9, r11 -> r16
...
```

- Code space savings can be made even on small patterns
  - In the given example we could save 1 instruction in 8 – a saving of 12.5%

```
...
   call [r10](r3,r6) -> r9
   call absfunc(r1,r2) -> r20,r17
   div r20, r17 -> r19
...
...
   mul r1, r2 -> r7
   call absfunc(r3,r5) -> r16,r13
...
```

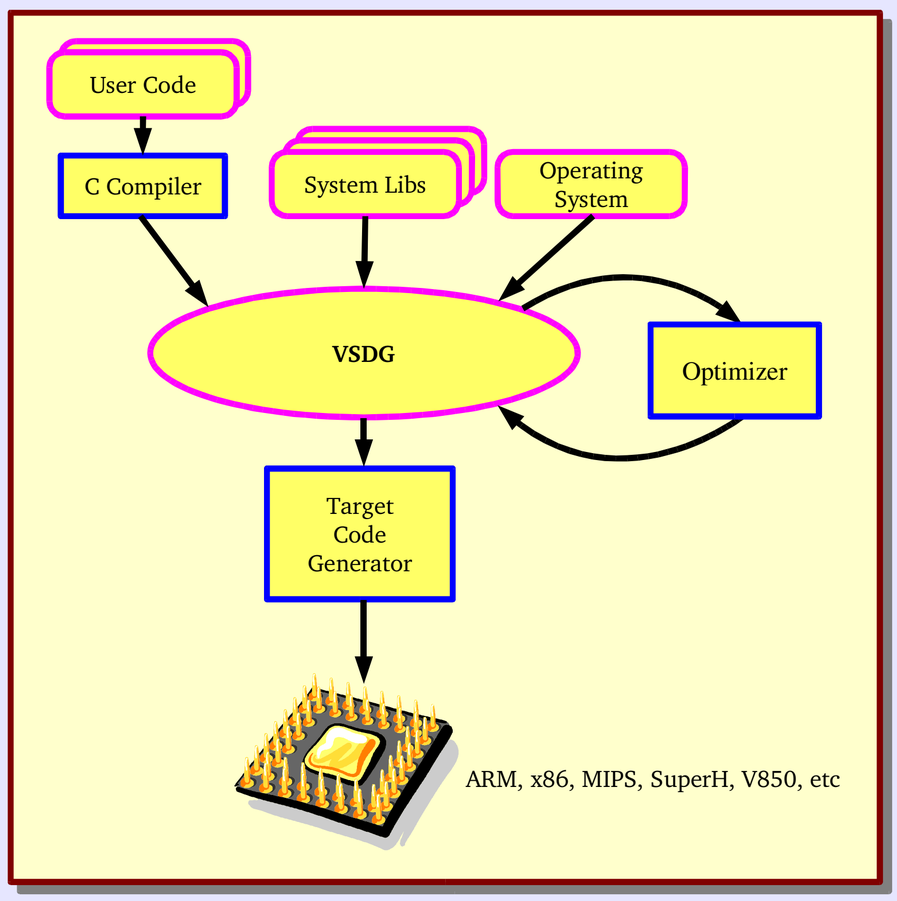
```
proc absfunc(2),2:
  add r1, r2 -> r3
  sub r1, r2 -> r4
  add r3, r4 -> r5
  mul r1, r2 -> r6
  ret r5, r6
end
```

COMPACTION vs. COMPRESSION

- **Compacted code is directly executable** and can therefore be debugged in the usual way.
  - Compressed code requires extra software to decompress the target code into an executable form prior to execution. Industry requires directly executable target code for quality assurance.
- **No additional system resources are needed** to run a compacted program.
  - Compressed code requires decompression into a runtime buffer before it can be executed by the processor, or a bytecode interpreter to execute the compressed bytecode stream.
- **Minimal time penalty for calling abstract functions**.
  - Compressed code must either be decompressed on-the-fly or at start-up. Both entail high time penalty, especially at higher levels of compression (*cf.* ZIP compression algorithms).
- **The compacted program is analysable and executable at all levels**.
  - Compression converts the program code into an opaque block of data whose behaviour cannot be directly inferred.

THE APPROACH TAKEN

- Compile whole program into common intermediate representation – ***tri*VM**
  http://www.trivm.net
- **Apply Whole-Program Optimization** on the program graph:
  - Targets new program graph: **Value-State Dependence Graph (*VSDG*)**
  - Currently exploring **Procedural Abstraction** as a means to factorize code body
  - **Classical optimizations** can also be applied (both control-flow and data-flow-based)
  - **We include as much code as possible** – application, operating system, drivers, etc
  - **More code leads to greater opportunities** for spotting duplicate code patterns
- Compilation flow:

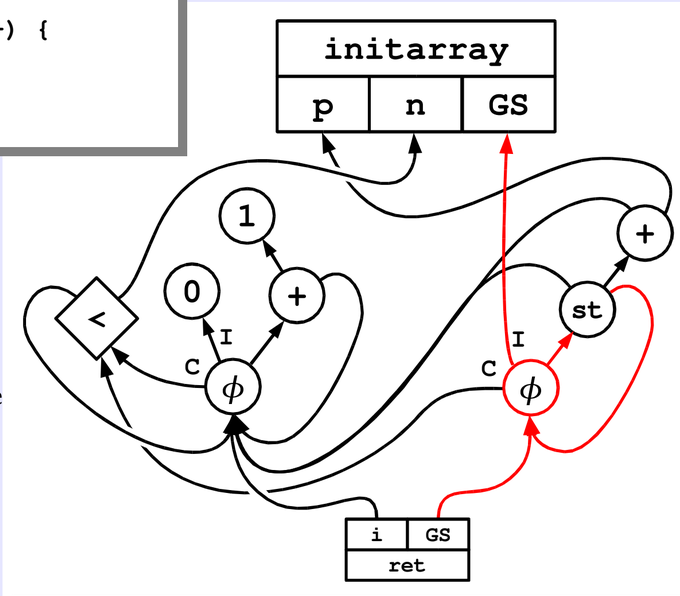


THE VALUE-STATE DEPENDENCE GRAPH

- Enhances the **Value Dependence Graph [Weise:1994]** with state dependency edges.
- Elegantly unifies both Control Flow Graph (CFG) and Data Flow Graph (DFG) in one common **SSA-based [Cytron:1990]** structure.

```
int initarray(int *p, int n) {
  int i;

   for(i=0; i<n; i++) {
      p[i] = i;
   }
   return i;
}
```



- **Key:-**
  - **C** - selector edge (control)
  - **I** - loop injector edge
  - **GS** – global state entry
  - Round nodes represent operations
  - Diamond node is a compare node (here, *less-than*)
  - Black edges are value-dep
  - Red edges are state-dep

This research was sponsored by a grant from ARM Ltd.
http://www.arm.com

For further details:
http://www.cl.cam.ac.uk/~nej22

UNIVERSITY OF CAMBRIDGE
Computer Laboratory