Studying the Applicability of the **Scratchpad Memory Management Unit**

Jack Whitham
Real-time Systems Group, University of York
jack@cs.york.ac.uk

Part I

Motivation and Background

Timing Composition

- What's the easiest way to compute timing properties of a complex embedded architecture?
  - ?

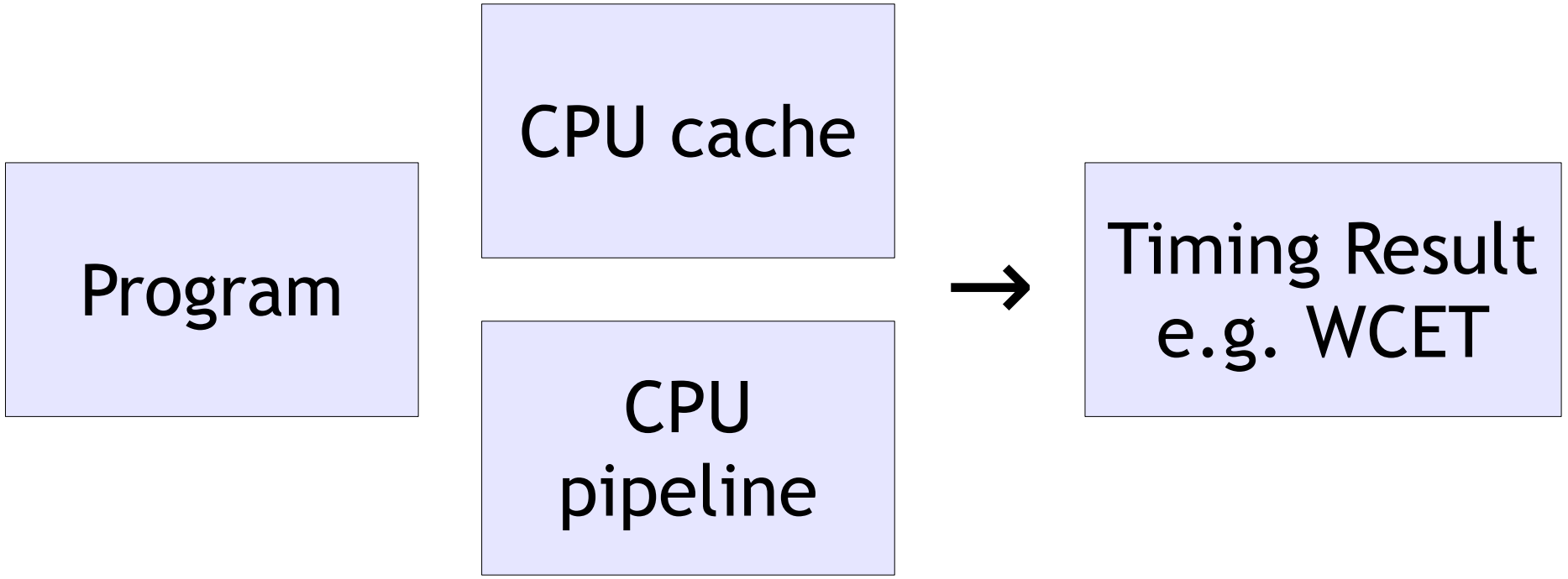
Timing Composition

- What's the easiest way to compute timing properties of a complex embedded architecture?
  - Divide and conquer.

- The *global* timing properties of a *system* (program & architecture) are determined by adding together separate *local* analyses for each of its components.

- A *timing-compositional* architecture enables this.

Example



- Timing analyses are carried out separately for each component, then added to obtain the timing result.

Timing-Compositional CPUs

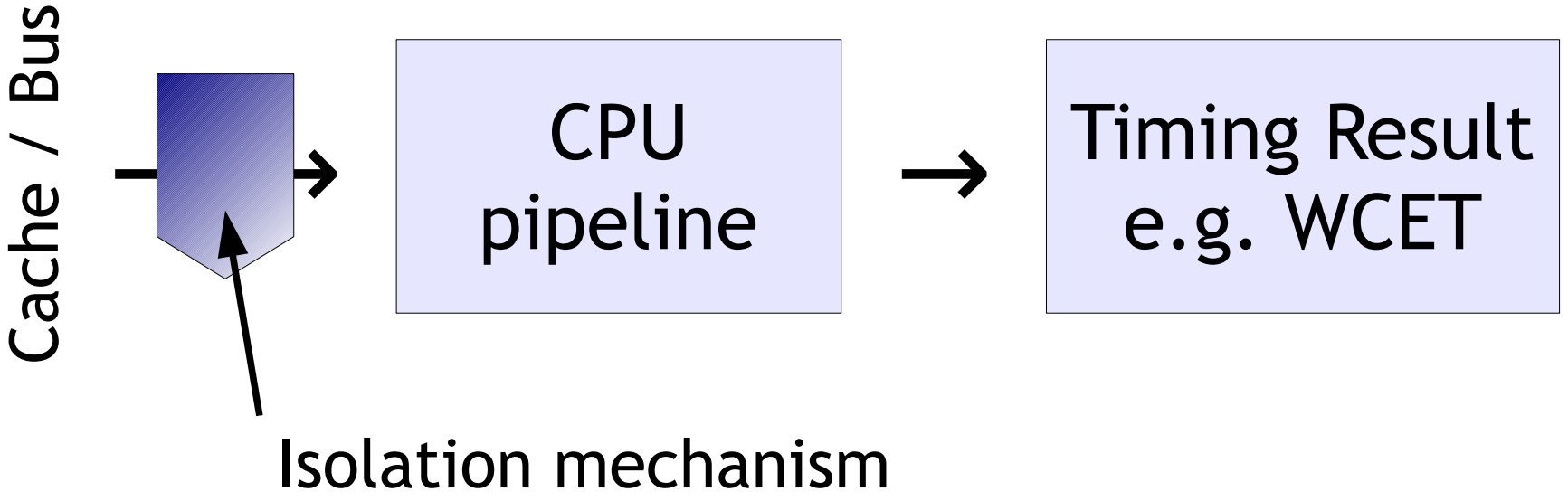
- Not all components have the required properties.
- CPUs that do are typically simple:
  - 68010, ARM7TDMI, JOP, PRET...
- Essential features for a t-c CPU:
  - A *timing accident* (e.g. cache miss) generated elsewhere does not change the internal state.
  - No *timing anomalies* are generated internally.
  - Limited throughput?

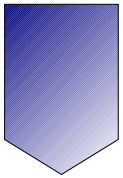
CPU Pipelines, in general

- Not timing-compositional:
  - Deep pipeline, multiple issue, out-of-order...
  - Timing anomalies may be generated internally.
  - Timing accidents occurring elsewhere will change the internal state.
- But some complex pipelines can be safely modeled:
  - Virtual traces.
  - CheckerMode.
  - Single-path pipelines.
- How can these approaches be timing-compositional?

“Conditional Compositionality”

- For complex pipelines, safe analysis requires isolation from timing accidents elsewhere:
  - Cannot freeze the entire pipeline instantaneously.
  - Any timing uncertainty is unacceptable.





Isolation Mechanism

- On-chip RAM - provides time-predictable memory access with zero uncertainty:
  - scratchpad memory (SPM) or locked cache.

- Suggested ideal *isolation property*:
  - Each *static instruction* has a fixed execution time that is easily determined by analysis.
  - Sufficient (arguably not *necessary*).

Ideal On-Chip RAM?

- Low-latency access to the *working set*.
- Each *static instruction* has a fixed execution time, easily determined by analysis. (Isolation property.)
- No restrictions on the working set: any object can be relocated on-chip for fast access.
- Programmer-friendly!

Data Cache Example

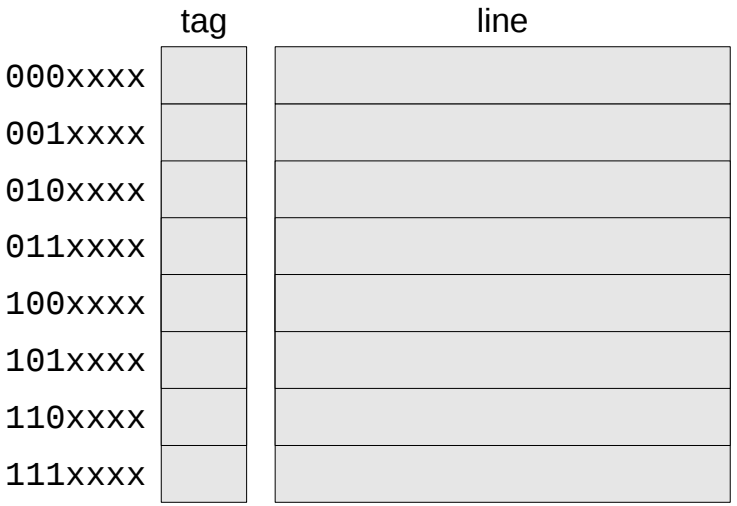
- Low-latency access to the *working set*. **Yes.**
- Each *static instruction* has a fixed execution time, easily determined by analysis. **No.** (No isolation.)
- No restrictions on the working set: any object can be relocated on-chip for fast access. **Yes.**
- Programmer-friendly! **Yes.**

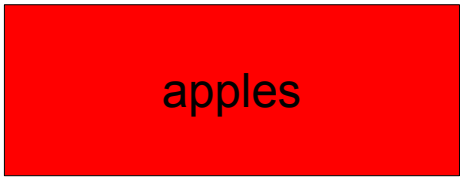
Lockable Data Cache

- Low-latency access to the *working set*. **Yes.**
- Each *static instruction* has a fixed execution time, easily determined by analysis. **Yes.**
- No restrictions on the working set: any object can be relocated on-chip for fast access. **No.**
- Programmer-friendly! **Maybe.**

Lockable DC Restriction

- Consider two objects and a simple data cache:



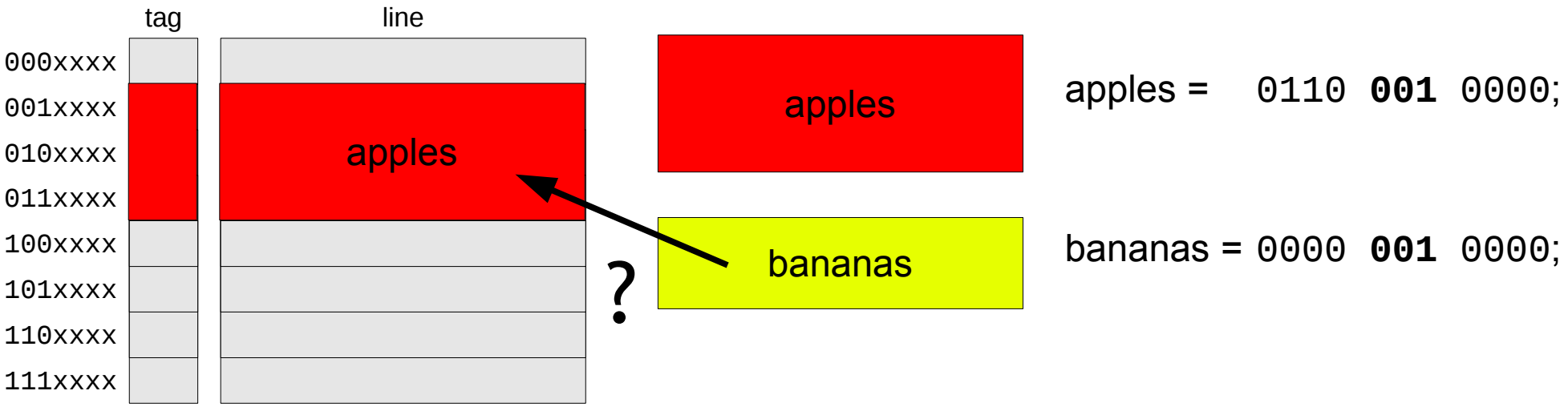


apples = 0110 **001** 0000;

bananas = 0000 **001** 0000;

Lockable DC Restriction

- In general, can't load & lock *N*+1 objects together in an *N*-way associative cache



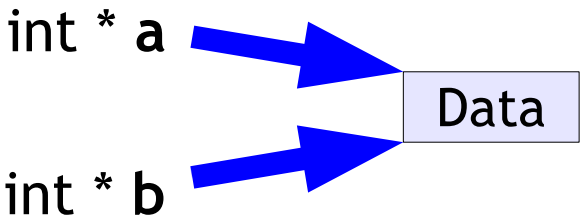
- Major restriction on the working set!

Scratchpad Memory (SPM)

- Low-latency access to the *working set*. **Yes.**
- Each *static instruction* has a fixed execution time, easily determined by analysis. **Yes.**
- No restrictions on the working set: any object can be relocated on-chip for fast access. **No.**
- Programmer-friendly! **Maybe.**

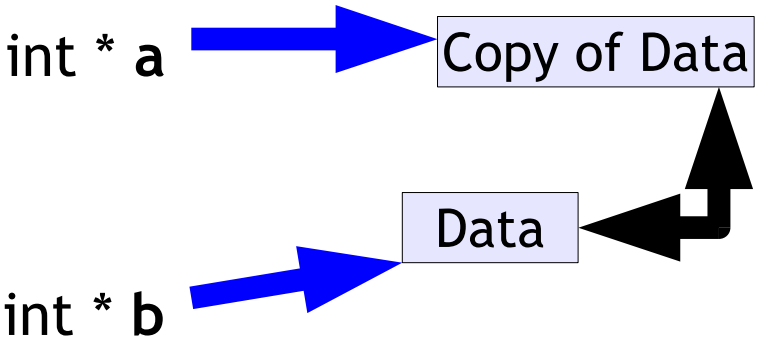
SPM Restriction

- If a program moves an object between external memory and SPM, its *address* changes.



before

a[1]=2;
b[1]=3;
(a[1] == 3) is **TRUE**



after

a[1]=2;
b[1]=3;
(a[1] == 3) is **FALSE**

SPM Restriction

- In general: relocating objects creates *pointer aliasing* and *pointer invalidation* problems.

- Most SPM research concentrates on domains where this is not an issue:
  - Restrict support to instruction memory
  - Restrict support to local & global variables
  - i.e. Restrictions on the working set.

- Dominguez et al. came up with a smart solution, but it loses the isolation property.

Scratchpad Memory Management Unit (SMMU)

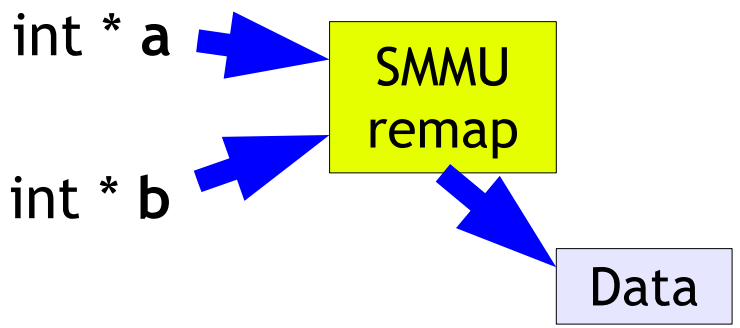
- Low-latency access to the *working set*. **Yes.**
- Each *static instruction* has a fixed execution time, easily determined by analysis. **Yes.**
- No restrictions on the working set: any object can be relocated on-chip for fast access. **Yes.**
- Programmer-friendly! **Maybe :)**

Address Transparency

- Objects can be relocated between external memory and SPM, as in previous work.
- But the relocations also update a “remapping table”, so that an object in SPM is accessed *using its address in external memory*.
- Pointer aliasing and invalidation: not an issue.

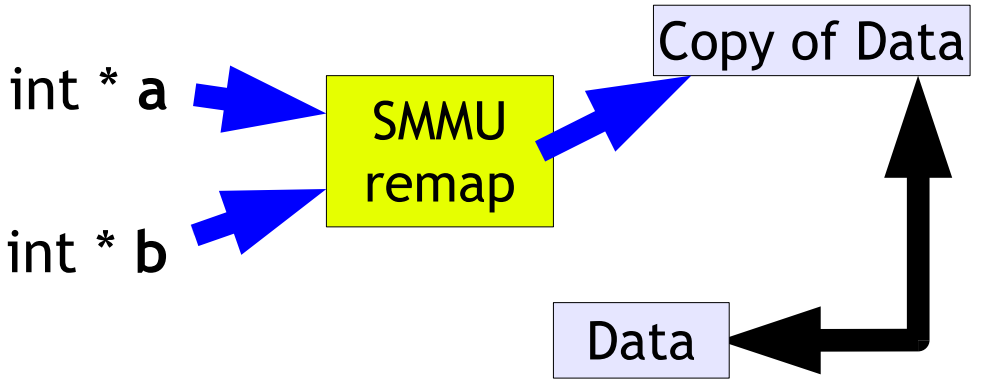
Address Transparency Example

- If a program moves an object between external memory and SPM, its *address does not change*.



before

a[1]=2;
b[1]=3;
(a[1] == 3) is **TRUE**



after

a[1]=2;
b[1]=3;
(a[1] == 3) is **TRUE**

Part II

Evaluation

Is the SMMU Useful?

- In theory, yes *(as discussed)*
- In case studies, yes *(see earlier publications)*
- But in general?
  - Suitable for all working sets *used in practice?*
  - Programmer friendly *in practice?*
  - That's the second half of this paper.

Allocation Algorithm

- How effective is automatic allocation of SPM space, when *any* data structure can be supported?
- Essential for “programmer friendliness”.
- Useful for evaluation: how well does the SMMU+SPM work in practice?

Evaluation Approach

- Take a set of benchmark programs.
  - Not necessarily real-time: intended as representative of programs in general.
- Use an algorithm to allocate SPM space for each program such that the estimated WCET is minimised.
- Identify the situations where the SPM couldn't be used, and find out why.

Allocation Algorithm

- Similar to Deverge and Puaut, in Proc. ECRTS'07.
- The control flow graph is *partitioned* at every entrance/exit from a loop.
- The *references to objects* used within each partition are determined by *def/use analysis*.
- A subset of these *references to objects* are chosen for the SPM.
- The main task of the algorithm is to choose the subsets such that the WCET is minimized.

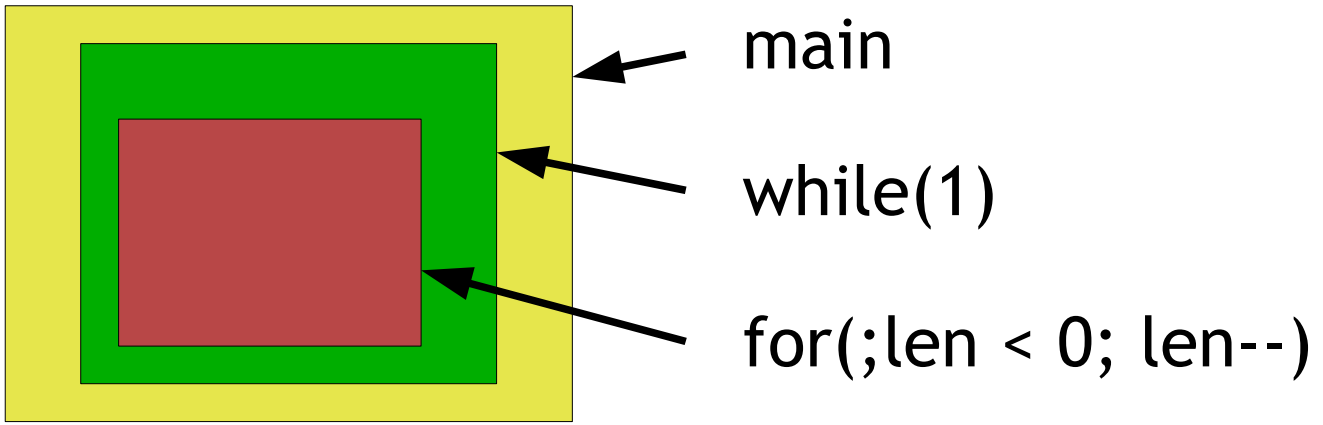
Example

```
void adpcm_coder(indata, outdata, len, state)
{
    outp = (signed char *)outdata;
    inp = indata;
    /* ... */
    for ( ; len > 0 ; len-- ) {
        val = *inp++;
        /* ... */
        delta |= sign;
        index += indexTable[delta];     /* size: 16 */
        if ( index < 0 ) index = 0;
        if ( index > 88 ) index = 88;
        step = stepsizeTable[index];    /* size: 89 */

        if ( bufferstep ) {
            outputbuffer = (delta << 4) & 0xf0;
        } else {
            *outp++ = (delta & 0x0f) | outputbuffer;
        }
        bufferstep = !bufferstep;
    }
    /* ... */
}
```

Example

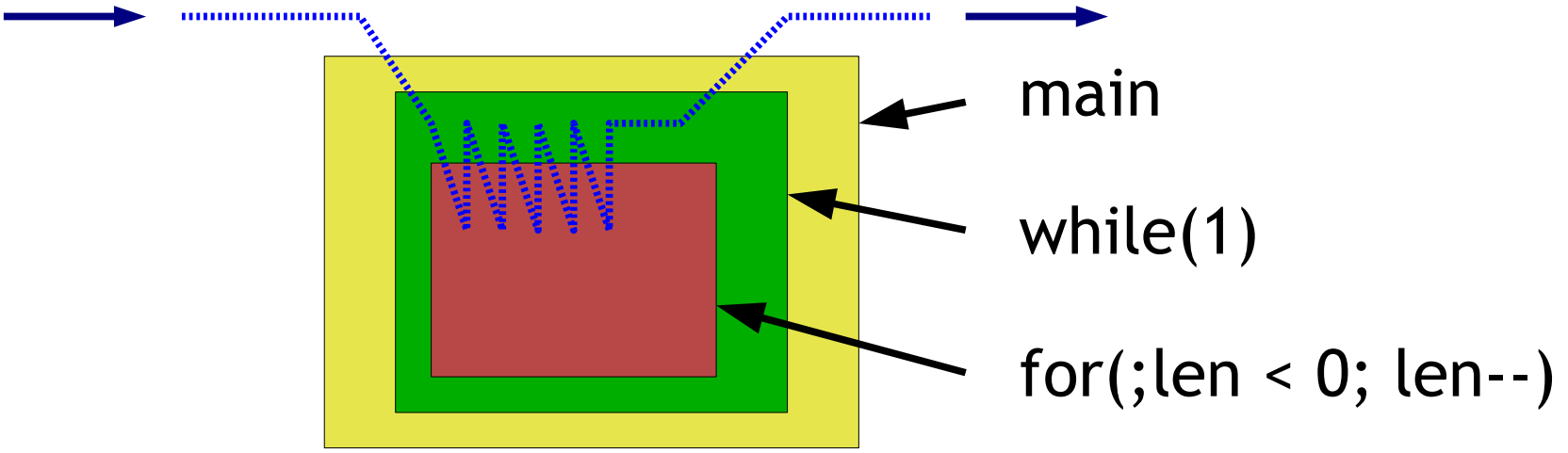
```
char    abuf[NSAMPLES/2];
short   sbuf[NSAMPLES];

main() {
    /* ... */
    while(1) {
        n = read(0, sbuf, NSAMPLES*2);
        if ( n == 0 ) break;
        adpcm_coder(sbuf, abuf, n/2, &state);
        write(1, abuf, n/4);
    }
    /* ... */
}
```

Regions:



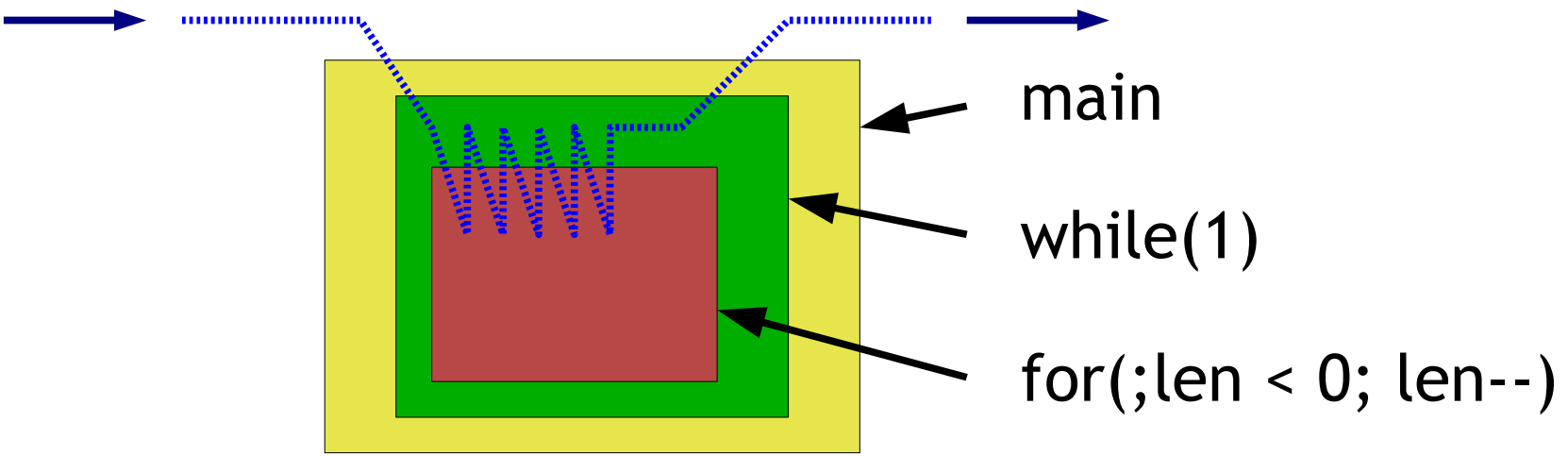
Reducing the WCET

- Method 1: move a reference to an object in region *R* into the SPM.
  - Benefit: accesses using that reference within *R* complete in 1 clock cycle instead of *L*.
  - Cost: each crossing of the boundary of *R* incurs a penalty for transferring the object.



Reducing the WCET

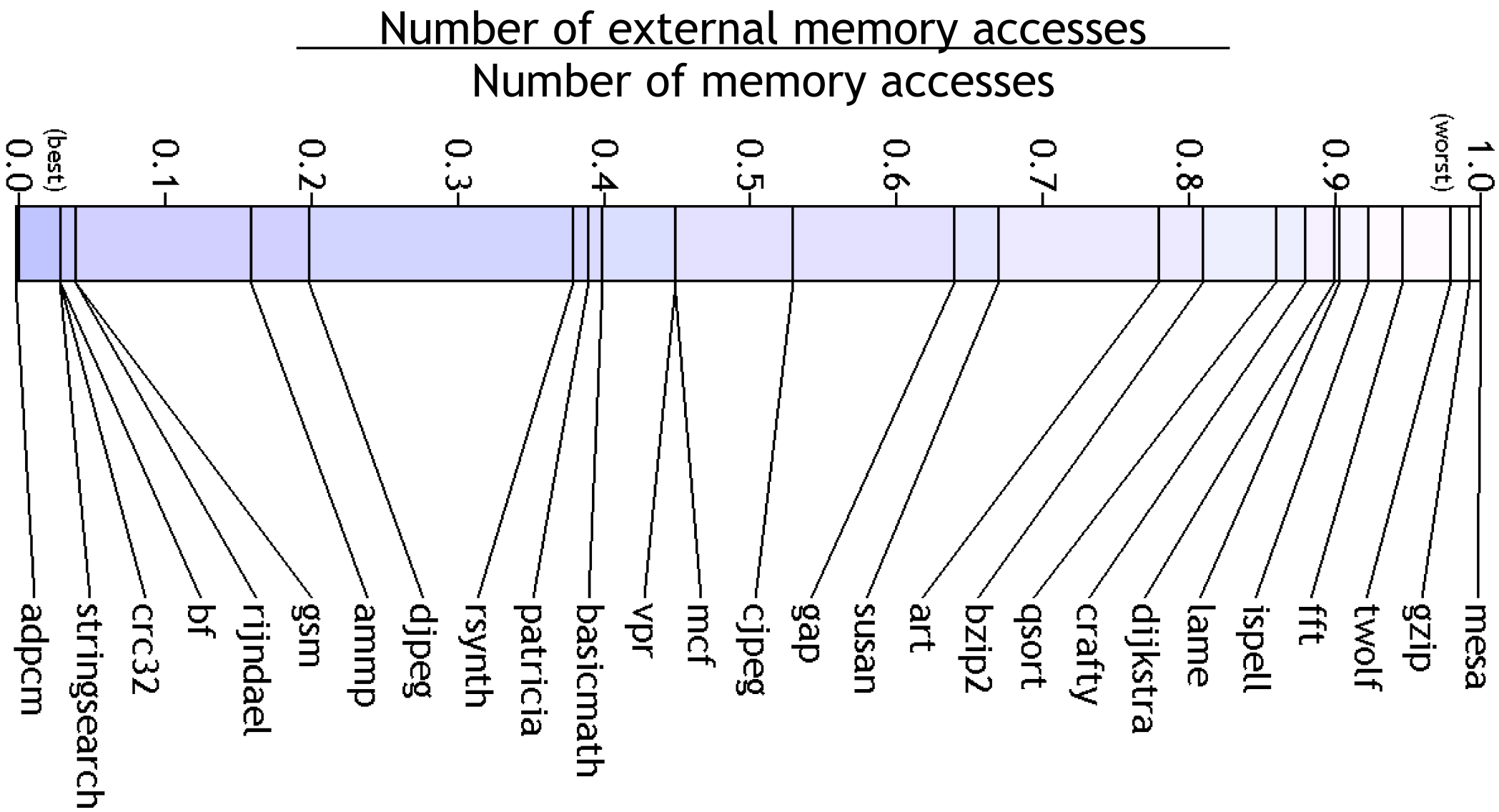
- Method 2: expand the set of regions that a particular reference to an object is in SPM.
  - Benefit: fewer of the total number of boundary crossings incur a penalty.
  - Cost: SPM space is in use for longer periods.



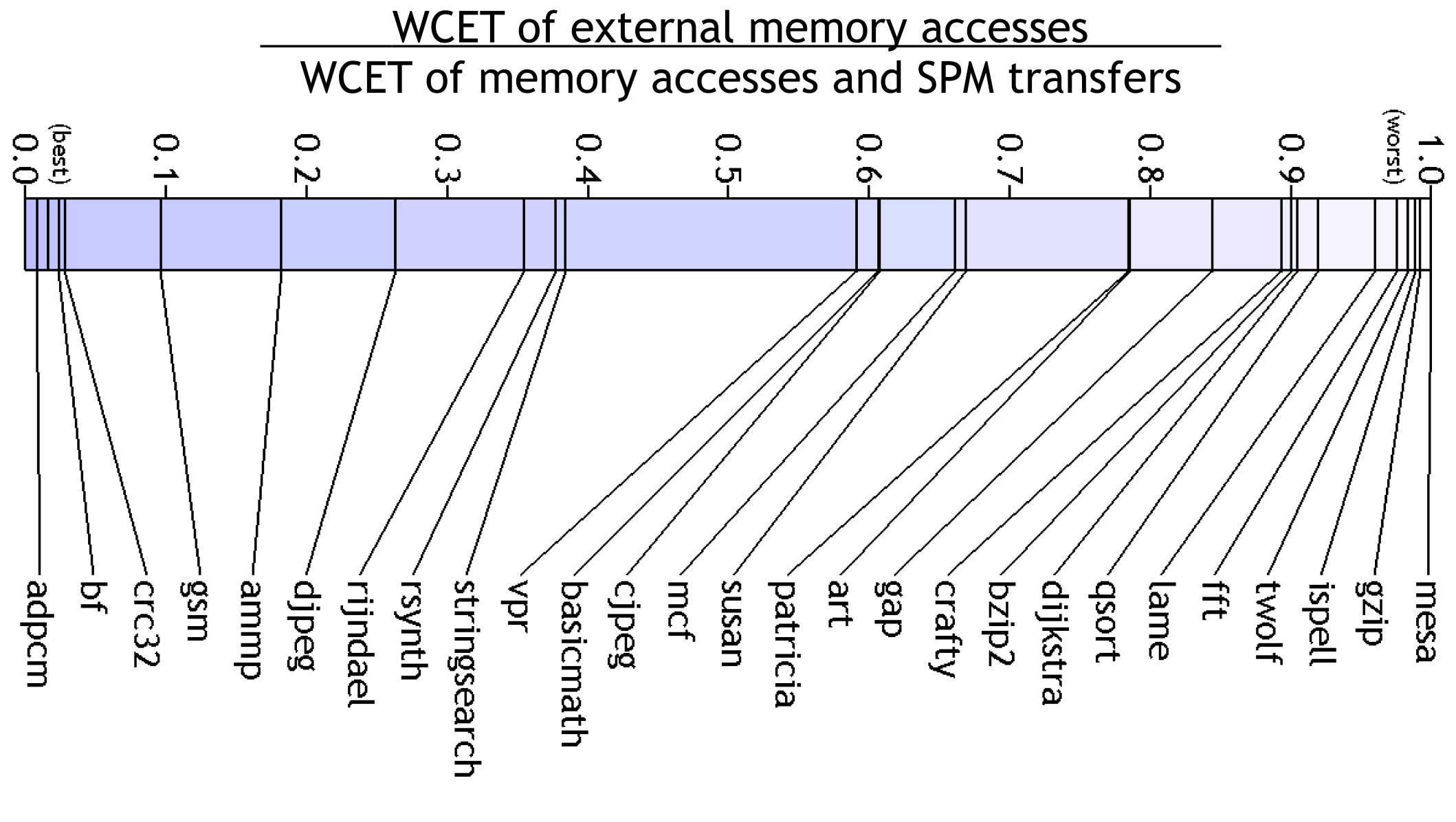
Statistics

- Although the SMMU can support any sort of data, the evaluation considered only accesses to dynamic data structures.
  - Where the address is neither decided at link-time, nor derived from the stack pointer.

Statistics


Number of external memory accesses
Number of memory accesses
0.0
(best)
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1.0
(worst)
adpcm
stringsearch
crc32
bf
rijndael
gsm
ammp
djpeg
rsynth
patricia
basicmath
vpr
mcf
cjpeg
gap
susan
art
bzip2
qsort
crafty
dijkstra
lame
ispell
fft
twolf
gzip
mesa

Statistics


WCET of external memory accesses
WCET of memory accesses and SPM transfers
0.0
(best)
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1.0
(worst)
adpcm
bf
crc32
gsm
ammp
djpeg
rijndael
rsynth
stringsearch
vpr
basicmath
cjpeg
mcf
susan
patricia
art
gap
crafty
bzip2
dijkstra
qsort
lame
fft
twolf
ispell
gzip
mesa

Comments

- Good results for some programs.
- But plenty of cases where the SPM did very little good... why?

Problem #1: Too Large

- Sometimes a benchmark will reference a very large object using a pointer.
  - rsynth: 74kb arrays
  - bzip2: 60Mb of data to be compressed
  - qsort: 1Mb to be sorted

- Too large to fit entirely in SPM. Accesses go direct to external memory.
- No method of “tiling” is currently considered.

Problem #2: No Benefit

- Sometimes the cost of transferring an object to SPM outweighs the time saved by using it.
  - equake: Happens for more than 75% of the object references
  - dijkstra: for over 50%.
- SPM transfers come in pairs: to/from.
- No optimization for “read only” references is currently used.

Problem #3: Poor Algorithms

- Sometimes, a better choice of algorithm or data structure would vastly reduce the memory accesses.
  - dijkstra benchmark: misuse of linked list.
  - art: matrix rows/cols should be transposed.

- A problem for data caches, too.
- A side effect of SMMU+SPM allocation is that poor *data locality* is revealed.

The Next Steps

- Find and evaluate solutions for the first two problems.
- Directly compare SMMU+SPM performance against a data cache.
- Investigate the possibility of adding SMMU+SPM allocation to a compiler.
- Consider multithreaded/multicore systems.

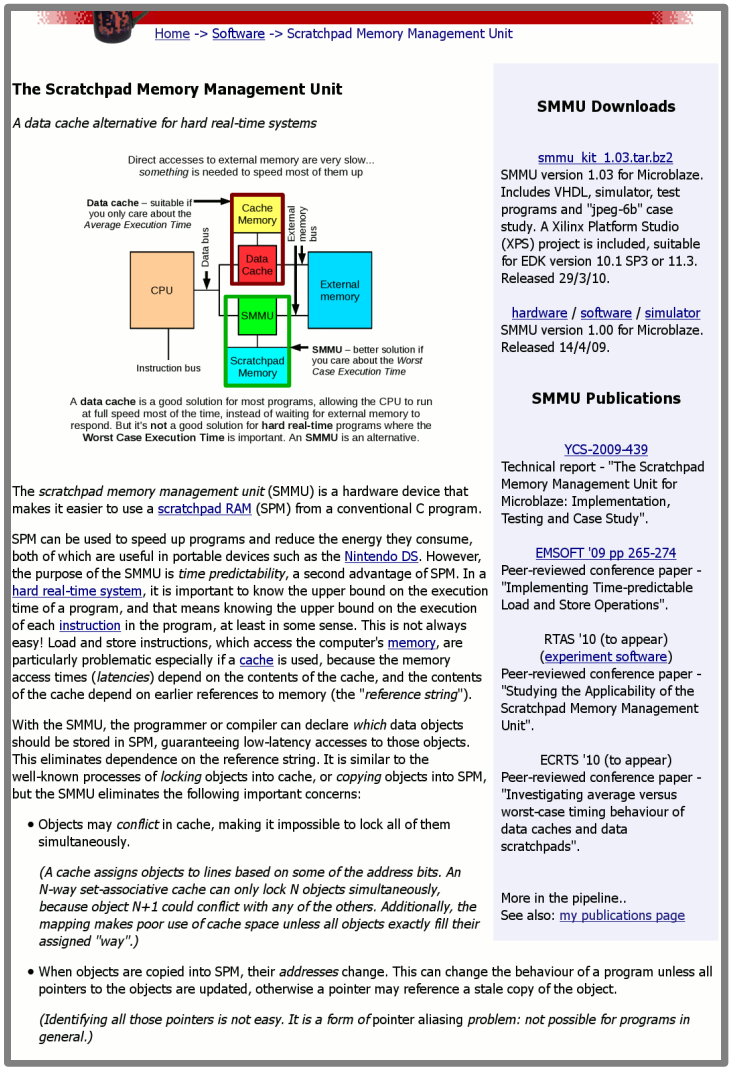
Conclusion

- Even these limited experiments have demonstrated that dynamic data structures can be stored in SPM:
  - Preserving the isolation property, and
  - Improving the WCET that can be determined by analysis.

- Plenty of opportunities for further exploration.

Thank you for your Attention

- Questions & comments most welcome, always.
- More information is on the www, search keywords: “SMMU, Whitham”



Home -> Software -> Scratchpad Memory Management Unit

The Scratchpad Memory Management Unit

A data cache alternative for hard real-time systems

Direct accesses to external memory are very slow... *something* is needed to speed most of them up

A **data cache** is a good solution for most programs, allowing the CPU to run at full speed most of the time, instead of waiting for external memory to respond. But it's **not** a good solution for **hard real-time** programs where the **Worst Case Execution Time** is important. An **SMMU** is an alternative.

The *scratchpad memory management unit* (SMMU) is a hardware device that makes it easier to use a scratchpad RAM (SPM) from a conventional C program.

SPM can be used to speed up programs and reduce the energy they consume, both of which are useful in portable devices such as the Nintendo DS. However, the purpose of the SMMU is *time predictability*, a second advantage of SPM. In a hard real-time system, it is important to know the upper bound on the execution time of a program, and that means knowing the upper bound on the execution of each instruction in the program, at least in some sense. This is not always easy! Load and store instructions, which access the computer's memory, are particularly problematic especially if a cache is used, because the memory access times (*latencies*) depend on the contents of the cache, and the contents of the cache depend on earlier references to memory (the "*reference string*").

With the SMMU, the programmer or compiler can declare *which* data objects should be stored in SPM, guaranteeing low-latency accesses to those objects. This eliminates dependence on the reference string. It is similar to the well-known processes of *locking* objects into cache, or *copying* objects into SPM, but the SMMU eliminates the following important concerns:

- Objects may *conflict* in cache, making it impossible to lock all of them simultaneously.

  (A cache assigns objects to lines based on some of the address bits. An N-way set-associative cache can only lock N objects simultaneously, because object N+1 could conflict with any of the others. Additionally, the mapping makes poor use of cache space unless all objects exactly fill their assigned "way".)

- When objects are copied into SPM, their *addresses* change. This can change the behaviour of a program unless all pointers to the objects are updated, otherwise a pointer may reference a stale copy of the object.

  (Identifying all those pointers is not easy. It is a form of pointer aliasing *problem: not possible for programs in general.)*

SMMU Downloads

smmu_kit_1.03.tar.bz2
SMMU version 1.03 for Microblaze. Includes VHDL, simulator, test programs and "jpeg-6b" case study. A Xilinx Platform Studio (XPS) project is included, suitable for EDK version 10.1 SP3 or 11.3. Released 29/3/10.

hardware / software / simulator
SMMU version 1.00 for Microblaze. Released 14/4/09.

SMMU Publications

YCS-2009-439
Technical report - "The Scratchpad Memory Management Unit for Microblaze: Implementation, Testing and Case Study".

EMSOFT '09 pp 265-274
Peer-reviewed conference paper - "Implementing Time-predictable Load and Store Operations".

RTAS '10 (to appear)
(experiment software)
Peer-reviewed conference paper - "Studying the Applicability of the Scratchpad Memory Management Unit".

ECRTS '10 (to appear)
Peer-reviewed conference paper - "Investigating average versus worst-case timing behaviour of data caches and data scratchpads".

More in the pipeline..
See also: my publications page

This slide intentionally left blank

(For the purposes of this talk, *ignore* instructions and instruction caching: analysis of instruction caches is a relatively well-understood problem.)

Why do we use Data Caches?

1. Improve Execution Time

Problem	Solution	Solved using a Cache?
Program execution time is bounded by off-chip memory access time		

1. Improve Execution Time

Problem	Solution	Solved using a Cache?
Program execution time is bounded by off-chip memory access time	Store working data set on-chip	**Yes**

System	Latency (CPU clock cycles)	CPU frequency (MHz)	Bus frequency (MHz)
ARM MPcore	**79**	210	70
StrongARM-110	**17**	50	50
PPC 405	**33**	100	125
Microblaze	**31**	125	125

2. Transparency to Programs

Problem	Solution	Solved using a Cache?
Program execution time is bounded by off-chip memory access time	Store working data set on-chip	**Yes**
Dynamic data structures and pointers must be supported		

2. Transparency to Programs

Problem	Solution	Solved using a Cache?
Program execution time is bounded by off-chip memory access time	Store working data set on-chip	**Yes**
Dynamic data structures and pointers must be supported	Logical address of each object must not change as data is moved on/off chip	**Yes**

3. Time-predictability?

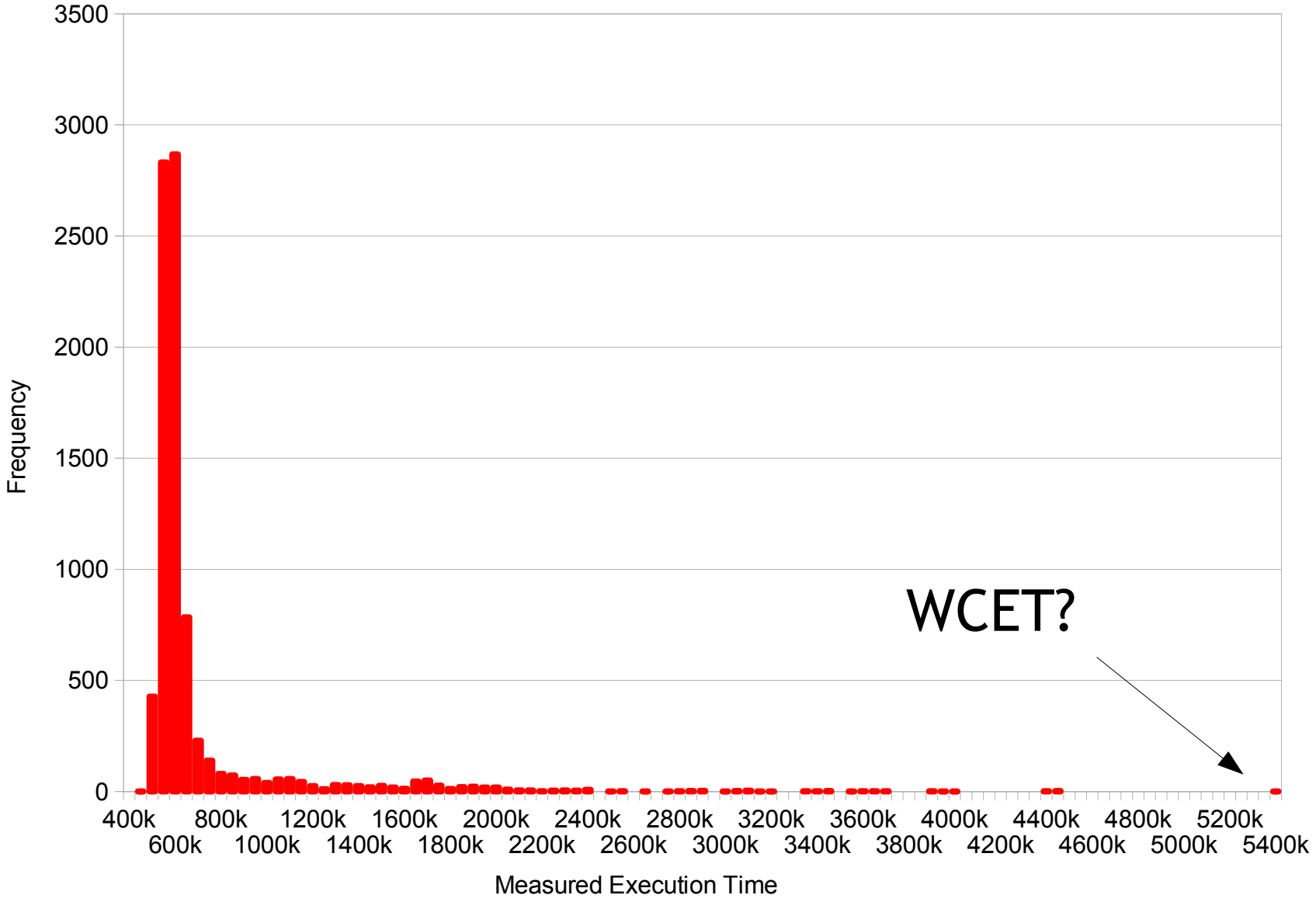
Problem	Solution	Solved using a Cache?
Program execution time is bounded by off-chip memory access time	Store working data set on-chip	**Yes**
Dynamic data structures and pointers must be supported	Logical address of each object must not change as data is moved on/off chip	**Yes**
Worst-case execution time needs to be calculated for a hard real-time system		

3. Time **un**predictability!

Problem	Solution	Solved using a Cache?
Program execution time is bounded by off-chip memory access time	Store working data set on-chip	**Yes**
Dynamic data structures and pointers must be supported	Logical address of each object must not change as data is moved on/off chip	**Yes**
Worst-case execution time needs to be calculated for a hard real-time system	Ensure that data access times are predictable during WCET analysis	**No**

Caches solve two problems *but create a third*.

Data Cache: arch-enemy of predictable systems



What's the Issue?

- The sequence of addresses used by a program is called its *reference string*.

- Depends on many things:

Array subscripts	x = random( ); `z = A[x];`
Base pointers	`A = malloc(...);`
Path through the code	`if(z) { y = A[x]; } else ...`

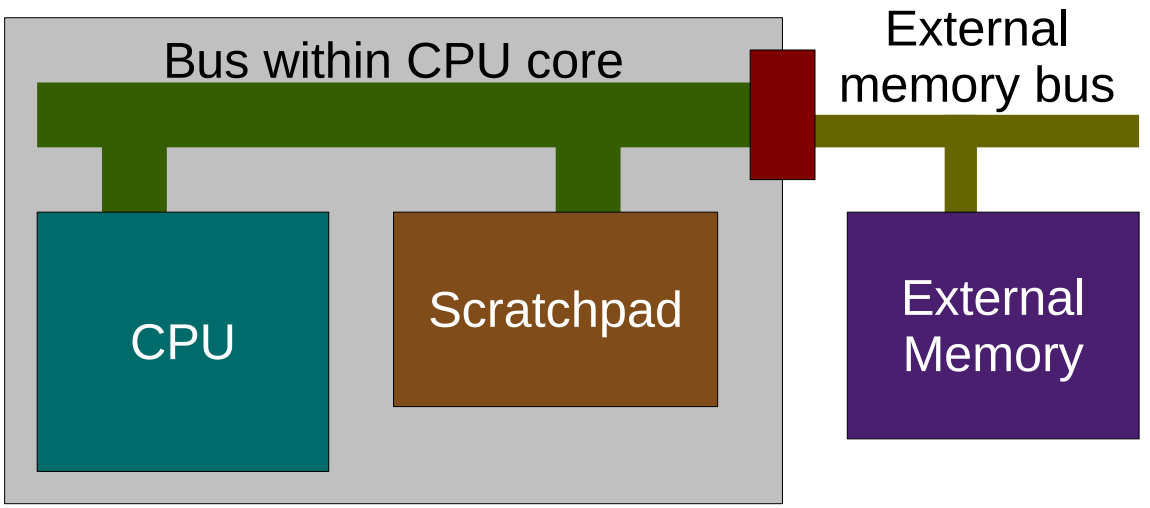
WCET Analysis?

- For any memory access operation X...
- The preceding reference string *defines the contents of the cache*.
- and therefore the *execution time* of X.

The execution time of each subprogram is potentially dependent on the behaviour of all other subprograms.

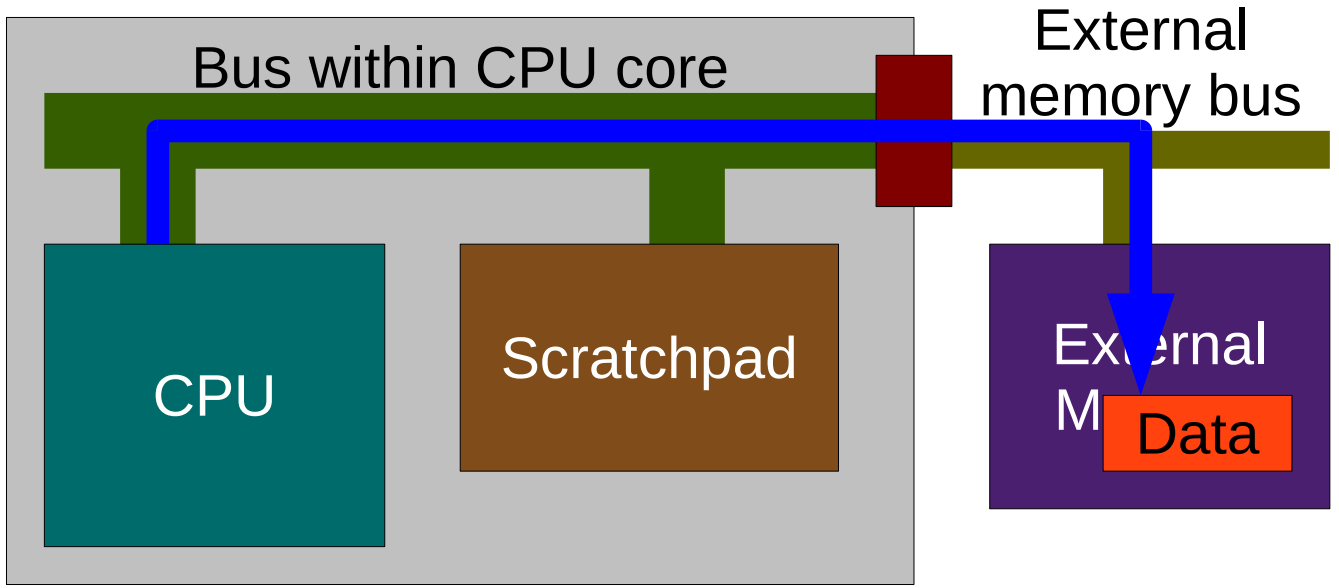
Possible Improvements

1. Scratchpad: a small, fast on-chip memory, which can be used to store commonly-accessed data.



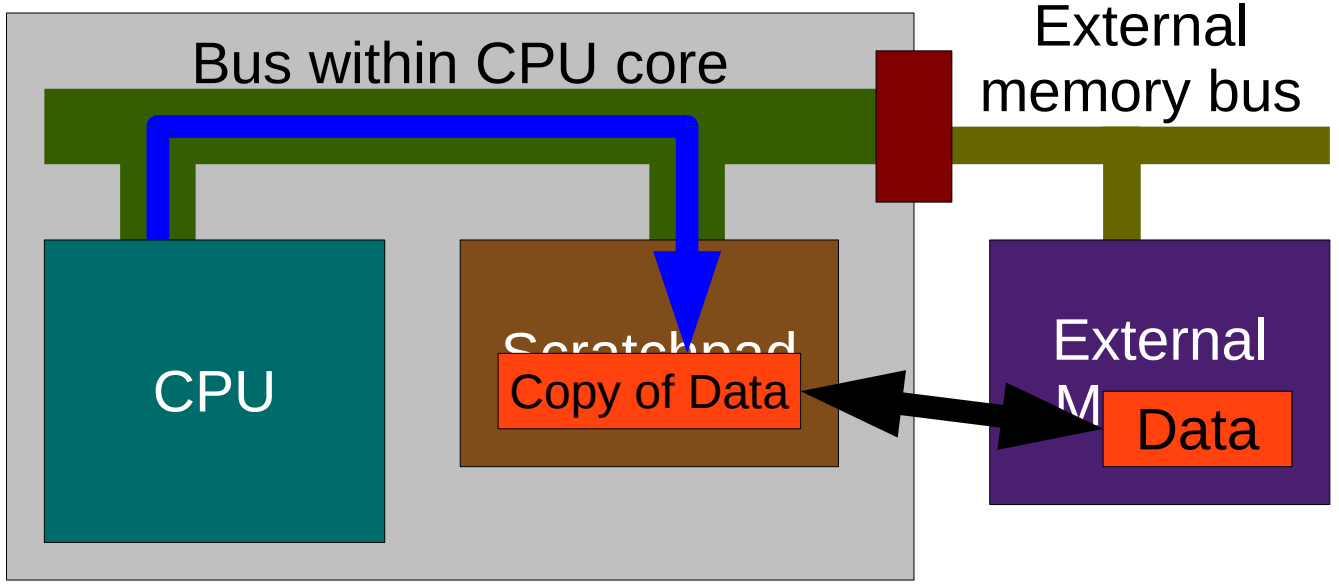
2. Cache locking: prevents any updates to all (or part) of the cache; the reference string has no effect.

Using a scratchpad



A cache *implicitly* relocates data to on-chip memory.

A scratchpad makes that process *explicit*, i.e. controlled by the program.

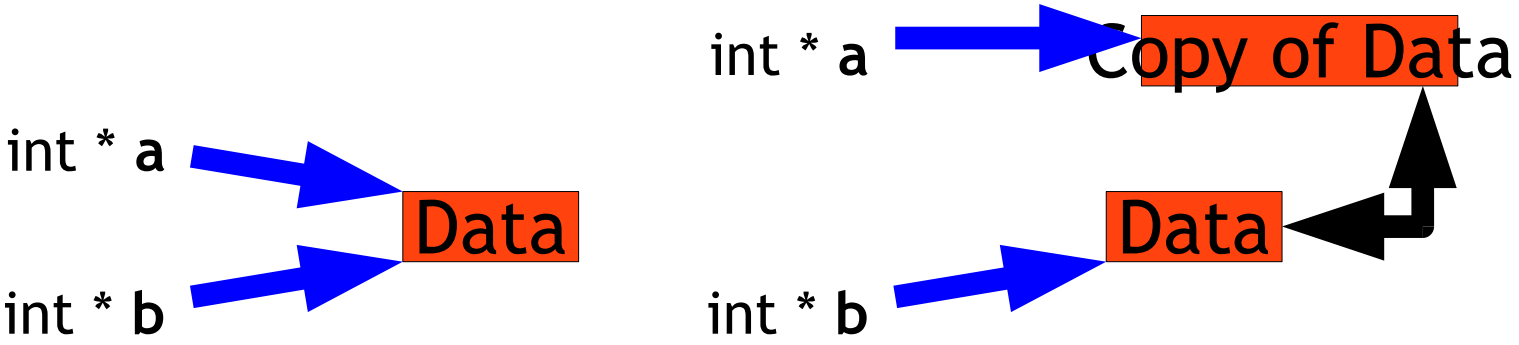


No dependence on reference string!

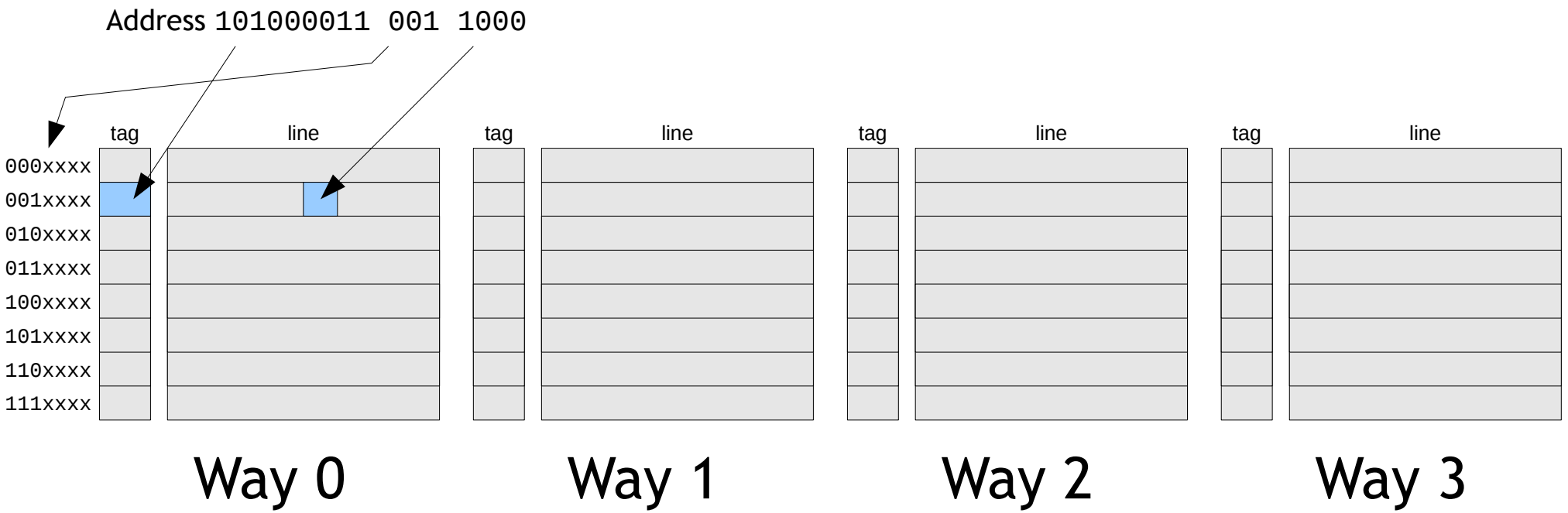
A problem

Problem	Solution	Solved using a Scratchpad?
Program execution time is bounded by off-chip memory access time	Store working data set on-chip	Yes
Dynamic data structures and pointers must be supported	Logical address of each object must not change as data is moved on/off chip	No
Worst-case execution time needs to be calculated for a hard real-time system	Ensure that data access times are predictable during WCET analysis	Yes

Physical relocation of data causes addresses to change. Issues are *pointer aliasing* and *pointer invalidation*: program semantics are affected!

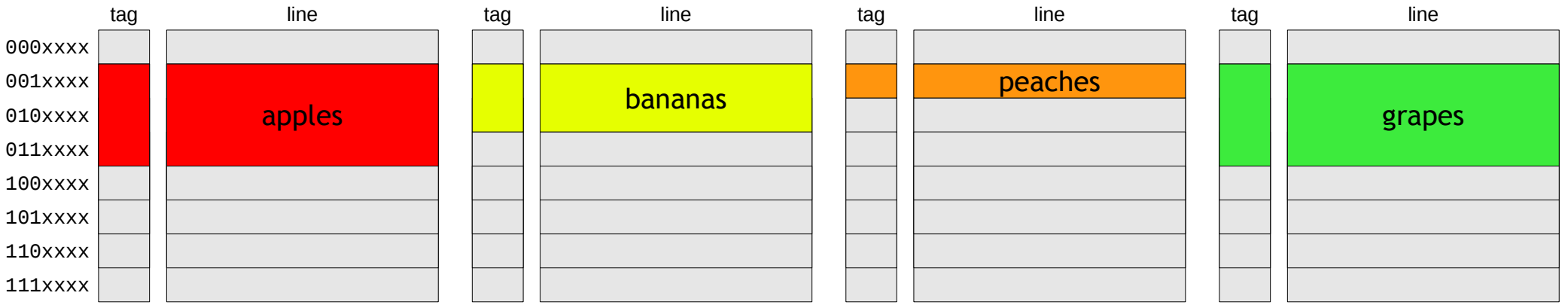


Using a locked cache



"Locking" disables the cache update mechanism: no dependence on the reference string.

Typical locking mechanisms:
entire cache, entire way, single line



Way 0 Way 1 Way 2 Way 3

apples

bananas

peaches

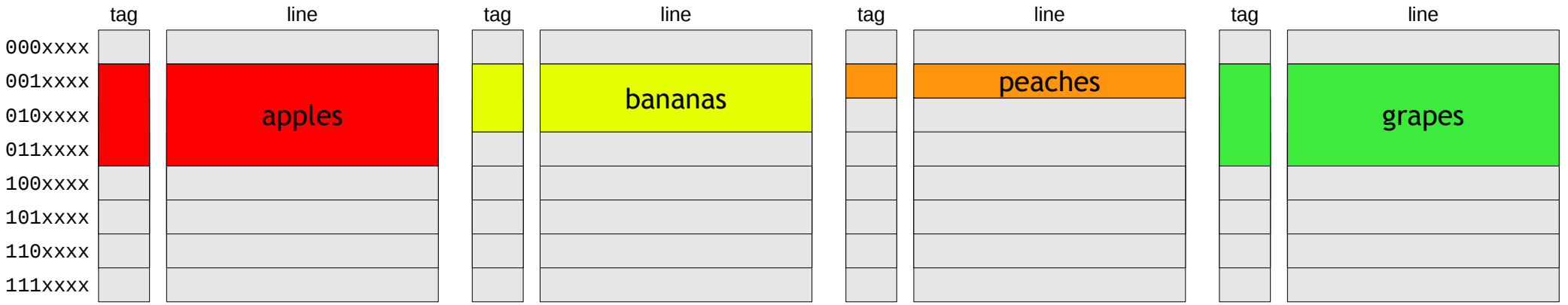
grapes

apples = `000100110 001 0000;`

bananas = `101100000 001 0000;`

peaches = `110101010 001 0000;`

grapes = `101000100 001 0000;`



apples

apples = `000100110 001 0000;`

bananas

bananas = `101100000 001 0000;`

peaches

peaches = `110101010 001 0000;`

grapes

grapes = `101000100 001 0000;`

plums

plums = `111110000 **001** 0000;`

Another problem

Problem	Solution	Solved using a Locked Cache?
Program execution time is bounded by off-chip memory access time	Store working data set on-chip	**Yes**
Dynamic data structures and pointers must be supported	Logical address of each object must not change as data is moved on/off chip	**Yes**
Worst-case execution time needs to be calculated for a hard real-time system	Ensure that data access times are predictable during WCET analysis	**Yes**
Locking mechanism depends on the address of each locked object	Allow objects to be relocated when locked	**No**

- For an *n*-way cache, a maximum of *n* objects can be safely locked *unless guarantees can be made about their addresses*.
- One-size-fits-all.

What (I think) we want

Problem	Solution	Solved using ???
Program execution time is bounded by off-chip memory access time	Store working data set on-chip	**Yes**
Dynamic data structures and pointers must be supported	Logical address of each object must not change as data is moved on/off chip	**Yes**
Worst-case execution time needs to be calculated for a hard real-time system	Ensure that data access times are predictable during WCET analysis	**Yes**
Locking mechanism depends on the address of each locked object	Allow objects to be relocated when locked	**Yes**

??? = hardware that replaces a cache.

Other solutions

- Don't use pointers or dynamic data structures:
  Common in present-day hard RTS.

- Use scratchpads only for temporary storage, not for existing objects – Wellings, Schoeberl

- *Shape analysis* and cache-aware memory allocation (CAMA) – Herter, Reineke, Wilhelm

- Don't *relocate* data: just *page* the scratchpad contents in and out – Barua et al.

Problem Analysis

Problem	Solution	Solved using ???
Program execution time is bounded by off-chip memory access time	Store working data set on-chip	**Yes**
Dynamic data structures and pointers must be supported	Logical address of each object must not change as data is moved on/off chip	**Yes**
Worst-case execution time needs to be calculated for a hard real-time system	Ensure that data access times are predictable during WCET analysis	**Yes**
Locking mechanism depends on the address of each locked object	Allow objects to be relocated when locked	**Yes**

+ Scratchpad memory is required

Problem	Solution	Solved using ???
Program execution time is bounded by off-chip memory access time	Store working data set on-chip	**Yes**
Dynamic data structures and pointers must be supported	Logical address of each object must not change as data is moved on/off chip	**Yes**
Worst-case execution time needs to be calculated for a hard real-time system	Ensure that data access times are predictable during WCET analysis	**Yes**
Locking mechanism depends on the address of each locked object	Allow objects to be relocated when locked	**Yes**

+ Program explicitly states which objects must be *locked into scratchpad* before their data is used
+ Any access to a locked object can be treated as a cache hit.
+ Any other access can be treated as a cache miss... but with no effect on the locked objects!

Problem	Solution	Solved using ???
Program execution time is bounded by off-chip memory access time	Store working data set on-chip	**Yes**
Dynamic data structures and pointers must be supported	Logical address of each object must not change as data is moved on/off chip	**Yes**
Worst-case execution time needs to be calculated for a hard real-time system	Ensure that data access times are predictable during WCET analysis	**Yes**
Locking mechanism depends on the address of each locked object	Allow objects to be relocated when locked	**Yes**

+ Hardware provides an address remapping f.

Physical address = f (Logical address)

+ When a locked object is accessed, an arbitrary offset is added to the logical address to obtain the physical address - in scratchpad!

- Scratchpad gains the transparency of a cache while retaining time-predictability.

Illustrating the concept

From three perspectives.

- The program;

- The hardware;

- The WCET analyser.

Illustration 1: the program

```
FOR i FROM 0 TO size-1 DO
   sum := sum + array[i];
END FOR;
```

```
x := start_of_list;
WHILE x <> NIL DO
   sum := sum + x.element;
   x := x.next;
END WHILE;
```

```
handle := OPEN(array, size, phys_addr);
FOR i FROM 0 TO size-1 DO
   sum := sum + array[i];
END FOR;
CLOSE(handle);
```

```
x := start_of_list;
WHILE x <> NIL DO
   sum := sum + x.element;
   x := x.next;
END WHILE;
```

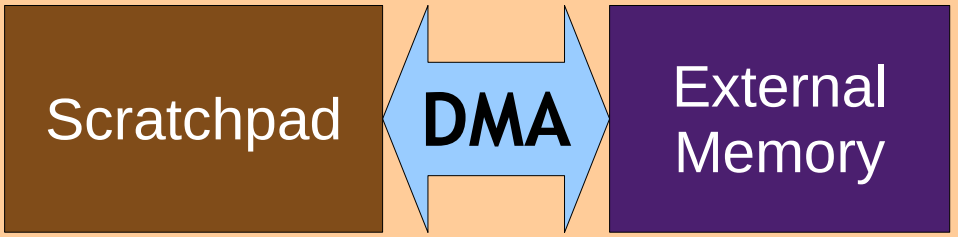
```
handle := OPEN(array, size, phys_addr);
FOR i FROM 0 TO size-1 DO
   sum := sum + array[i];
END FOR;
CLOSE(handle);
```

```
x := start_of_list;
WHILE x <> NIL DO
   handle := OPEN(x, SIZEOF(*x), phys_addr);
   sum := sum + x.element;
   x := x.next;
   CLOSE(handle);
END WHILE;
```

Illustration 2: the hardware

OPEN/CLOSE

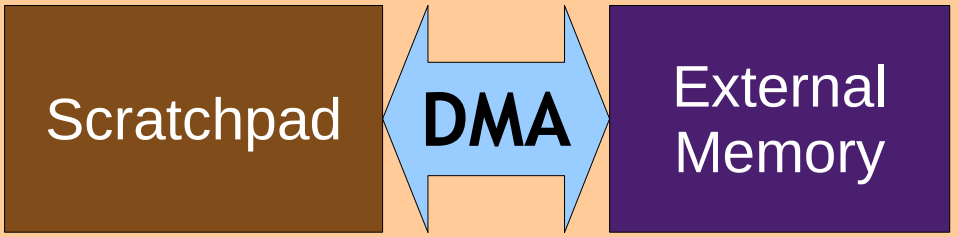
1.



2. Add/remove remapping table entry

```
handle :=
OPEN(array, size,
phys_addr);
```

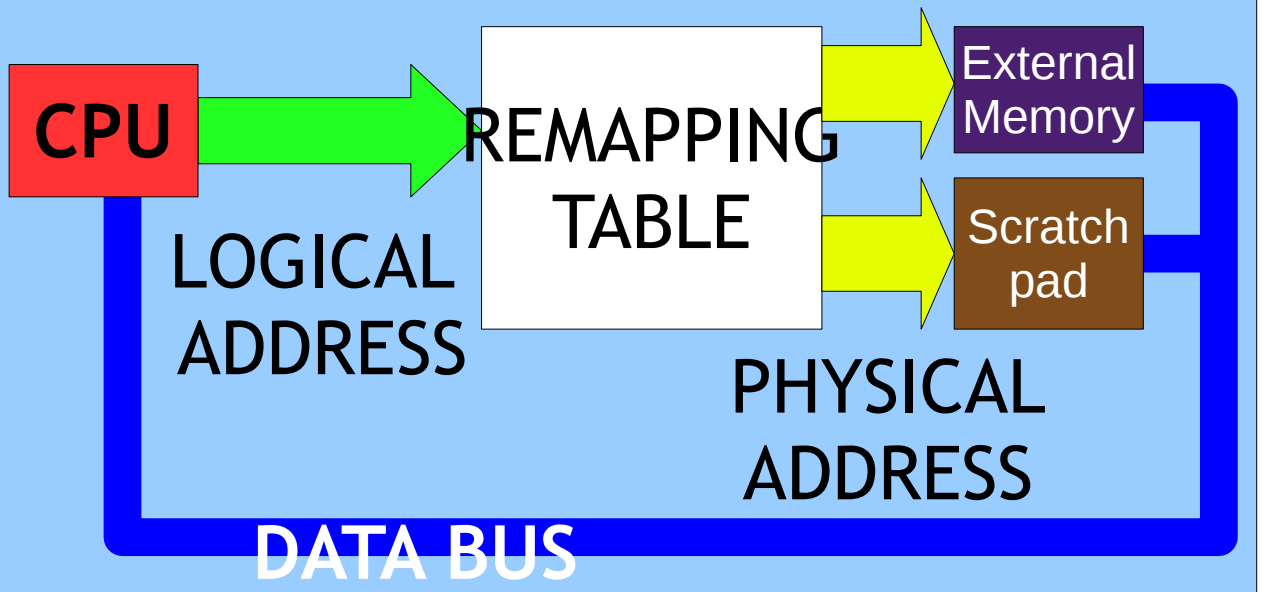
OPEN/CLOSE

1. 

2. Add/remove remapping table entry

```
handle :=
OPEN(array, size,
phys_addr);
```

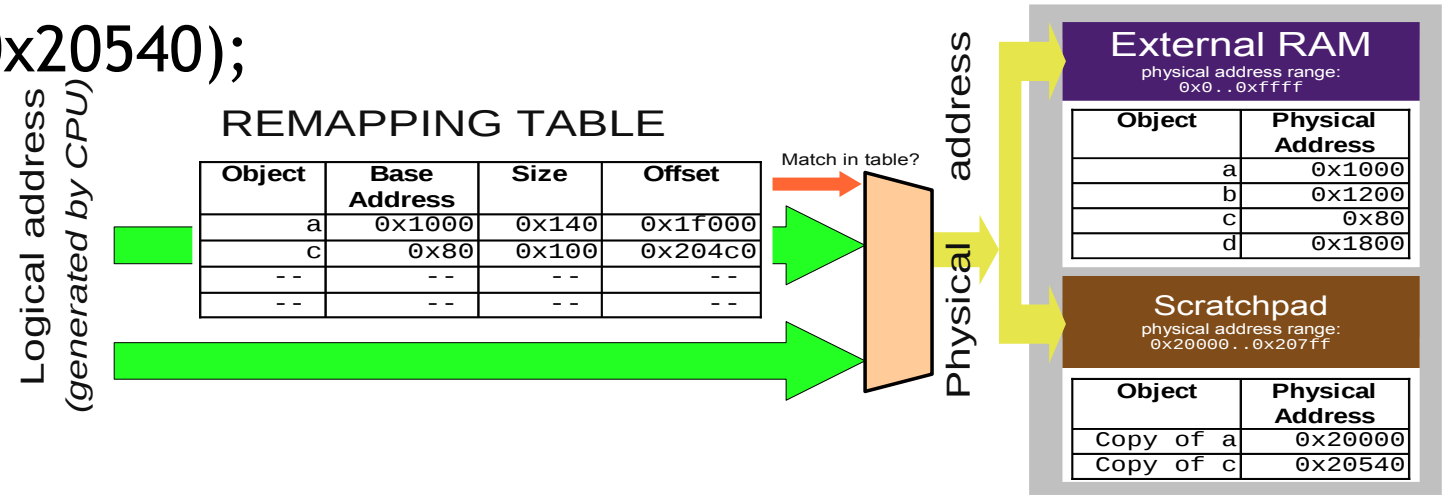
MEMORY ACCESS: LOAD/STORE




```
sum := sum +
array[i];
```

OPEN(a, 0x140, 0x20000);

OPEN(c, 0x100, 0x20540);



Object	Base Address	Size	Offset
a	0x1000	0x140	0x1f000
c	0x80	0x100	0x204c0
--	--	--	--
--	--	--	--

Object	Physical Address
a	0x1000
b	0x1200
c	0x80
d	0x1800

Object	Physical Address
Copy of a	0x20000
Copy of c	0x20540

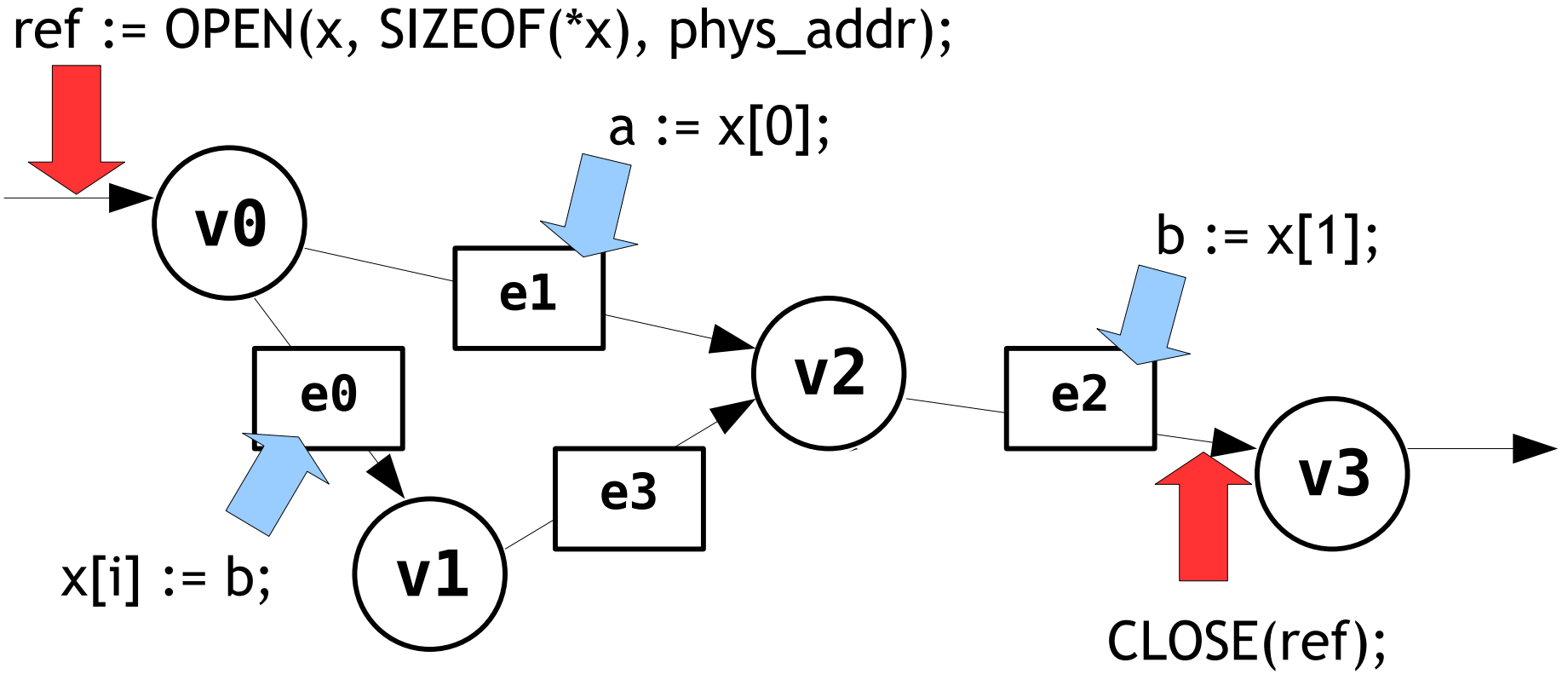
```
handle := OPEN(array, size, phys_addr);
FOR i FROM 0 TO size-1 DO
   sum := sum + array[i];
END FOR;
CLOSE(handle);
```

Memory access time
2 • DMA_Cost(size) + size

```
x := start_of_list;
WHILE x <> NIL DO
   handle := OPEN(x, SIZEOF(*x), phys_addr);
   sum := sum + x.element;
   x := x.next;
   CLOSE(handle);
END WHILE;
```

Memory access time
2 • length • DMA_Cost(SIZEOF(*x)) + 2

Illustration 3: the WCET analyser



Accesses to “x” are guaranteed to reference scratchpad: “guaranteed cache hits”

Generating physical addresses

- OPEN/CLOSE operations must be statically planned offline: e.g. by the programmer, during compilation, during post-compilation analysis.
  - Scratchpad space must be allocated in advance.
  - The size of objects being OPENed or CLOSEd must be known before execution.
  - Similar to loop bounds...

Weaknesses of the approach

Weaknesses of the Approach

- Data locality can only be exploited if it can be predicted offline.

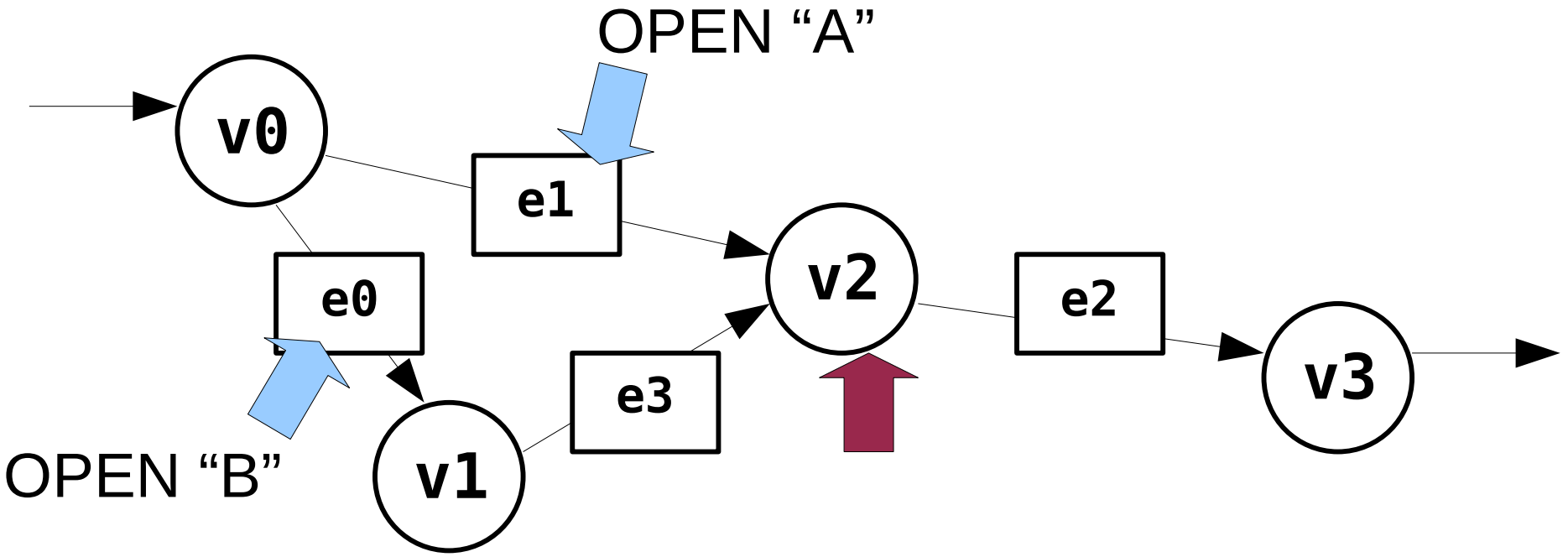
```
FOR x FROM 0 TO size-1 DO
   sum := sum + array[x];
END FOR;
```

```
FOR x FROM 1 TO count DO
   sum := sum + array[Random(x)];
END FOR;
```

```
x := start_of_list;
WHILE x <> NIL DO
   sum := sum + x.element;
   x := x.next;
END WHILE;
```

Weaknesses of the Approach

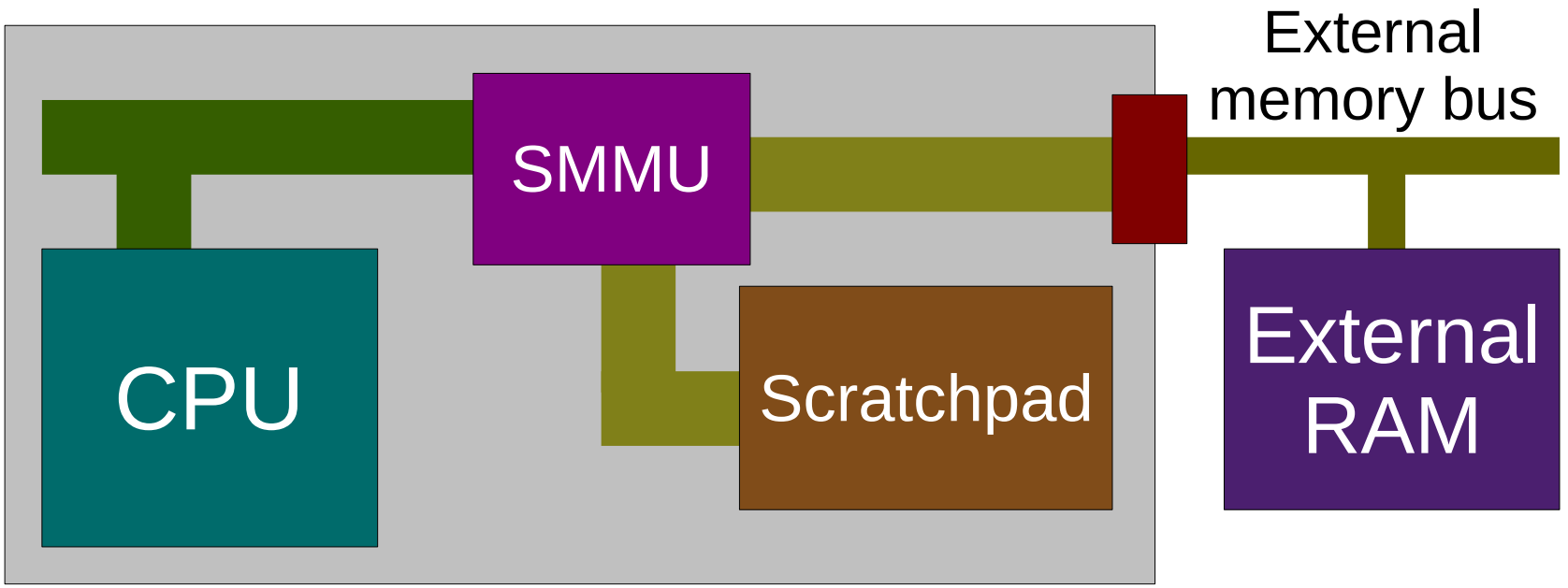
- The set of objects that are OPEN at any point in the program must be well-defined.



- But - still easier than considering abstract cache states because the number of possible states is so small, being unaffected by the reference string.

Implementation, Evaluation and Results

Scratchpad Memory Management Unit (SMMU)



SMMU: remapping table + DMA controller

Implemented on an FPGA and in a simulator

Experimental Assumptions

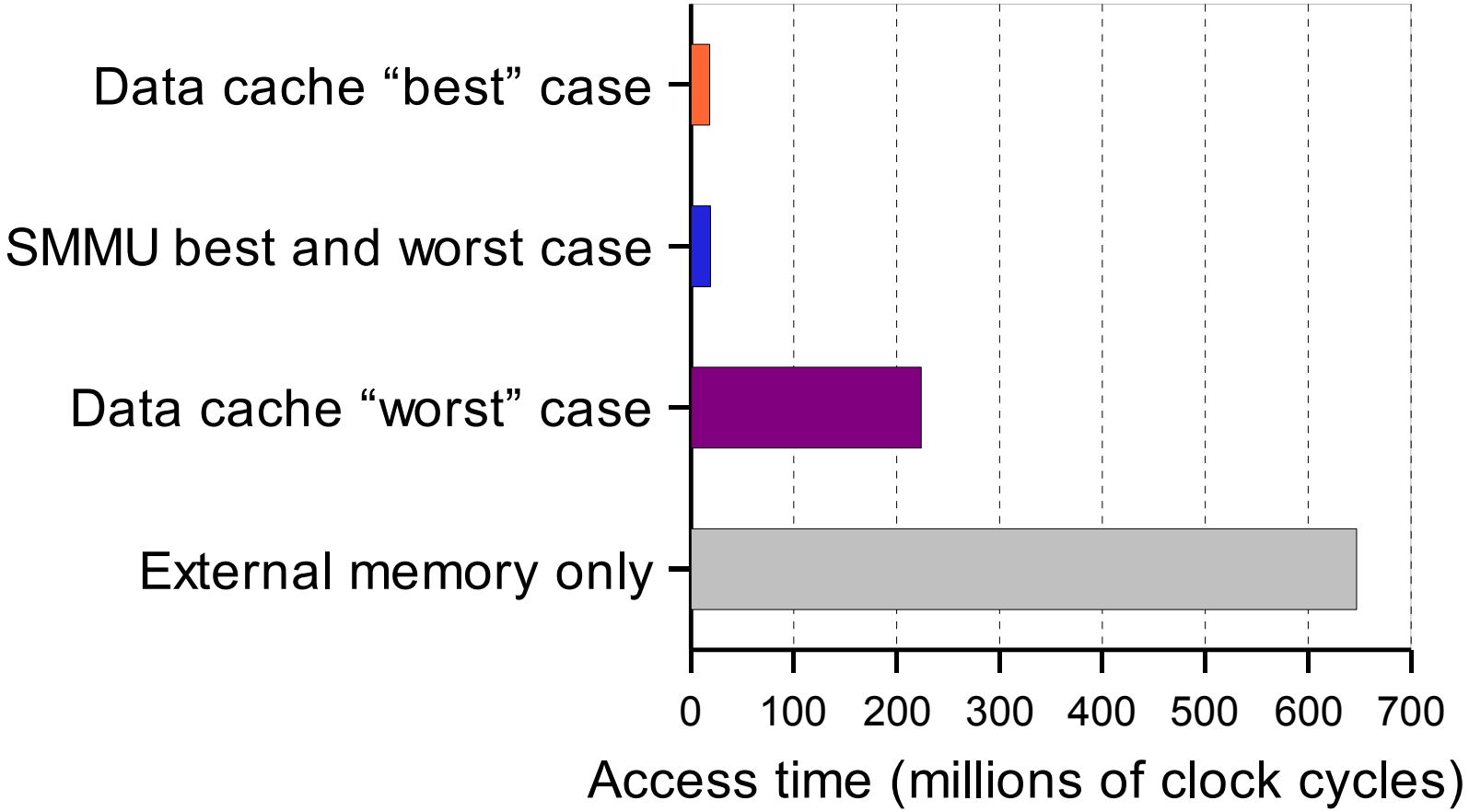
- Consider only accesses to dynamic data structures, i.e. LOAD/STORE operations with addresses that are unknown at compile time.

- Accesses to static data and instruction memory assumed handled by scratchpad.

- Single program, single thread, single CPU.

Initial Results

`ycc_rgb_convert` function (libjpeg) on a simulated platform:



Evaluation

- Evaluation really requires hard real-time benchmarks that use dynamic data structures.

- But those don't exist because of limitations of current WCET analysis.

- Solution: study conventional benchmark programs (SPEC, Mediabench, MIBench, etc.)

- Rationale: representative of real software.

- I adapted the non-real-time benchmarks to support pseudo-WCET analysis.

- I executed them once to capture:
  - control flow graph.
  - loop bounds (max. number of iterations).
  - the sizes of objects accessed by the code.

- Subsequent WCET analysis uses this data with no constraint on conditional statements.
  - not single-path analysis.

- I developed an algorithm to add SMMU operations OPEN and CLOSE to the programs.

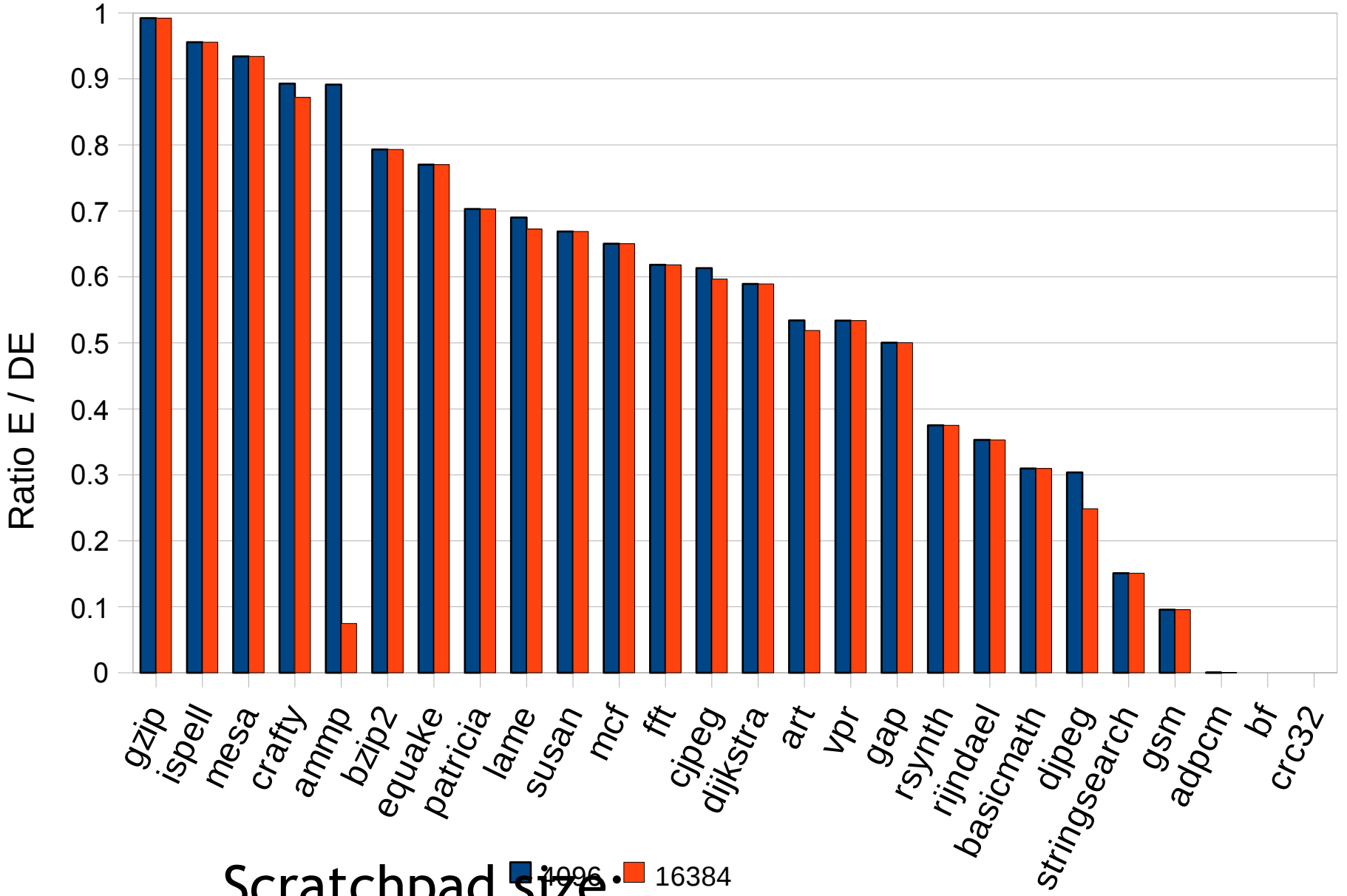
- The algorithm tries to minimise E:

Let E = WCET of accesses to dynamic data after the algorithm has done its work.

Let DE = baseline: maximum WCET of accesses to dynamic data - i.e. all go off chip.

E / DE = degree of WCET improvement achieved using the SMMU.

Results


Ratio E / DE
1
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
gzip
ispell
mesa
crafty
ammp
bzip2
equake
patricia
lame
susan
mcf
fft
cjpeg
dijkstra
art
vpr
gap
rsynth
rijndael
basicmath
djpeg
stringsearch
gsm
adpcm
bf
crc32
Scratchpad size:
4096
16384

Summary of Findings

- Quite often, objects are too large to be OPENed.

  gzip and bzip2 benchmarks in particular!

- The majority of memory accesses in each program are performed within a few “hot spots”

Summary of Findings

- Data locality can be a problem...
  but investigation revealed -

  1. The algorithms being used were sometimes poor,

     e.g. MIBench “Dijkstra”: iterate through a linked list to add one element to the tail!

  2. Or very inefficient with a cache as well,

     e.g. SPEC “Art”: matrix multiply iterates down a column instead of across a row.

  3. Or the code could be easily refactored to make OPEN and CLOSE more efficient.

     e.g. Mediabench “Lame”: one frequently-called procedure repeatedly dereferences the same pointer.

Conclusion

- Some problems are not solved by either caches or scratchpads.

- The SMMU is a possible solution.

- Investigation reveals that the SMMU's ability to reduce the WCET of a program is dependent on how it uses memory.

- Future work: examination and classification of memory-accessing hotspots in programs; loop tiling; paging; consider multi-CPU/multi-thread.

Thankyou

The Real-time Systems Group
at the University of York.
http://www.cs.york.ac.uk/rts/