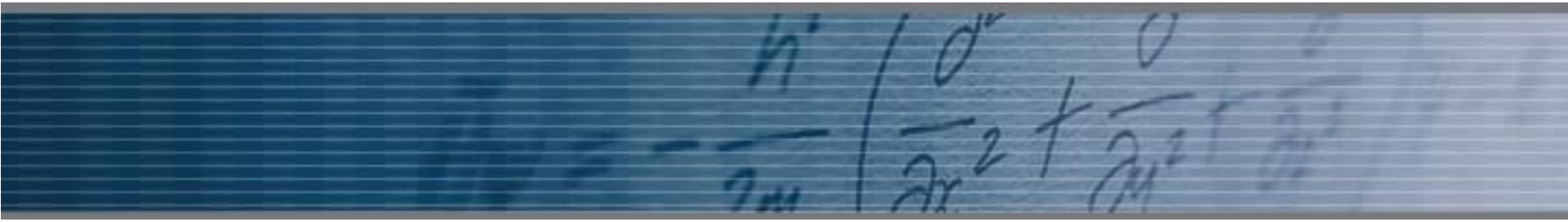


Program overlays revisited

Benedict R. Gaster

Outline

- Motivation
- Definition
- Implementation
- Evaluation
- Conclusion



A photograph of a chalkboard showing the time-independent Schrödinger equation. The equation is written in white chalk on a dark blue background. It is:
$$\hat{H}\psi = -\frac{\hbar^2}{2m} \left(\frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2} + \frac{\partial^2}{\partial z^2} \right) \psi = E\psi$$

Motivation

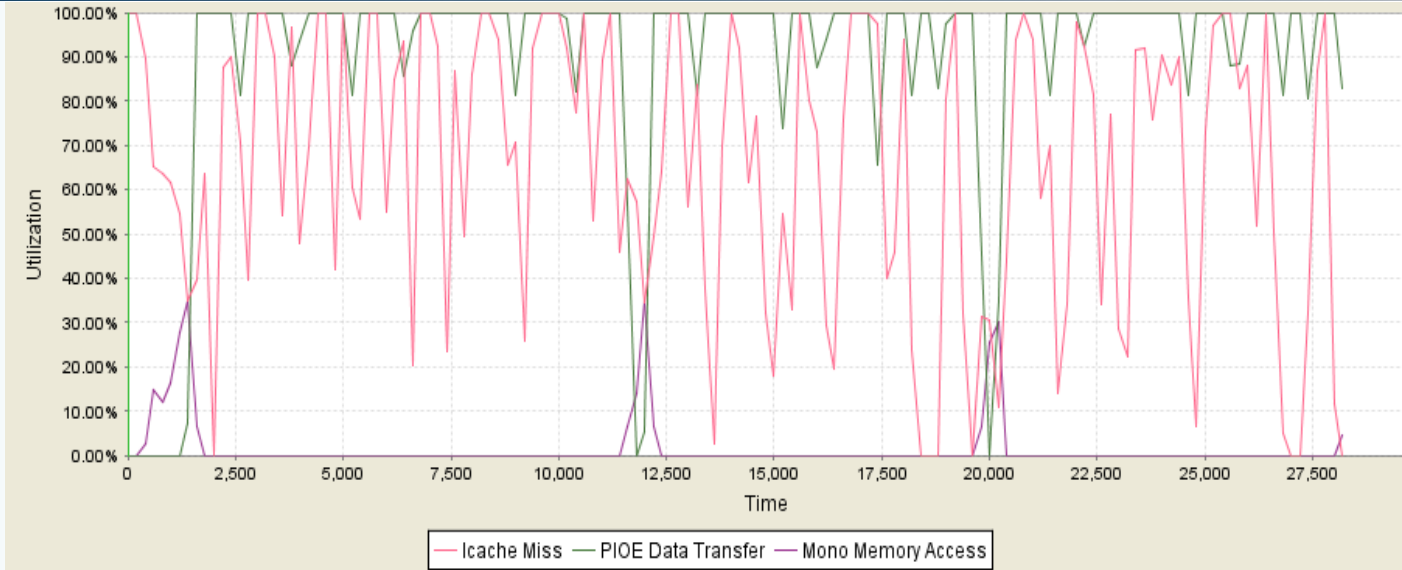
Motivation: Memory bandwidth contention

```
void perform_compute(double * poly p2m, poly double * res)
{
    int i = 0;
    for (i = 0; i < 3; i++)
    {
        async_memcpy2p(sem, res, p2m, sizeof(double) * 64);
        nops4k();
        sem_wait(sem);
    }
}
```

DRAM Requests

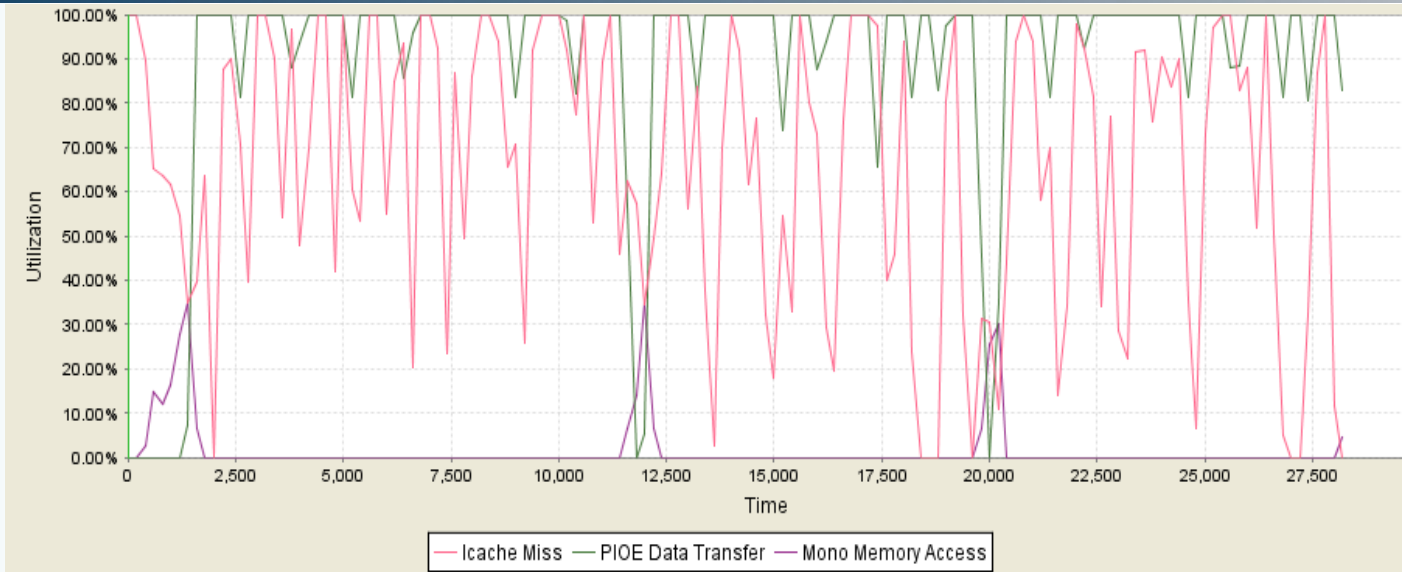
- Memory transfer
- Cache fill

Motivation: Memory bandwidth contention

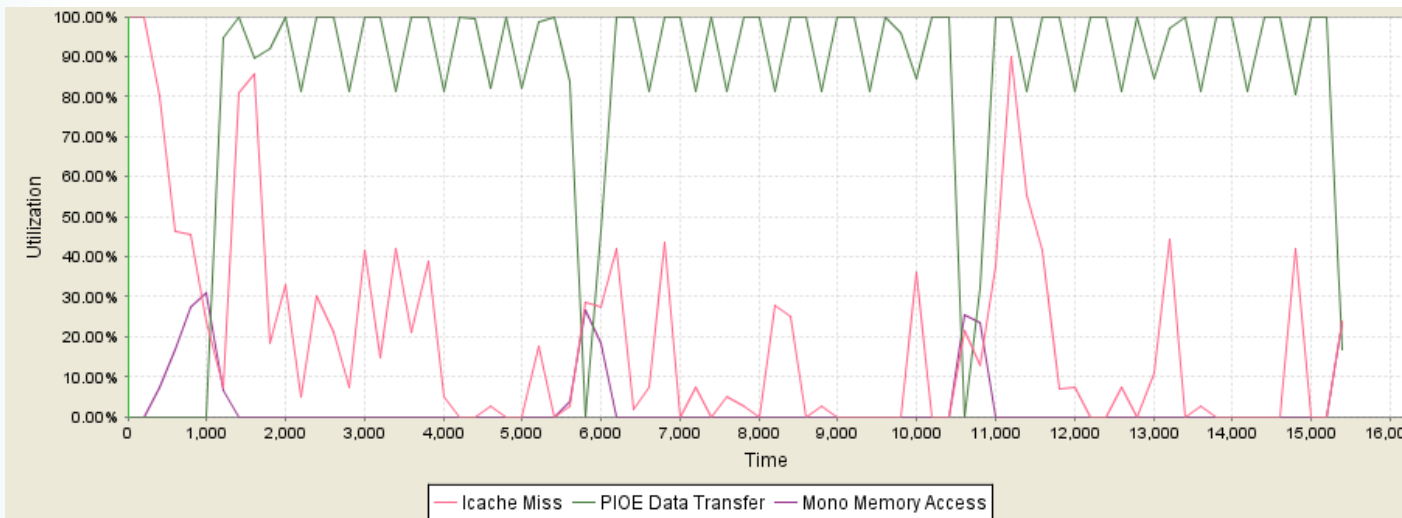


Code
run
from
DDR

Motivation: Memory bandwidth contention

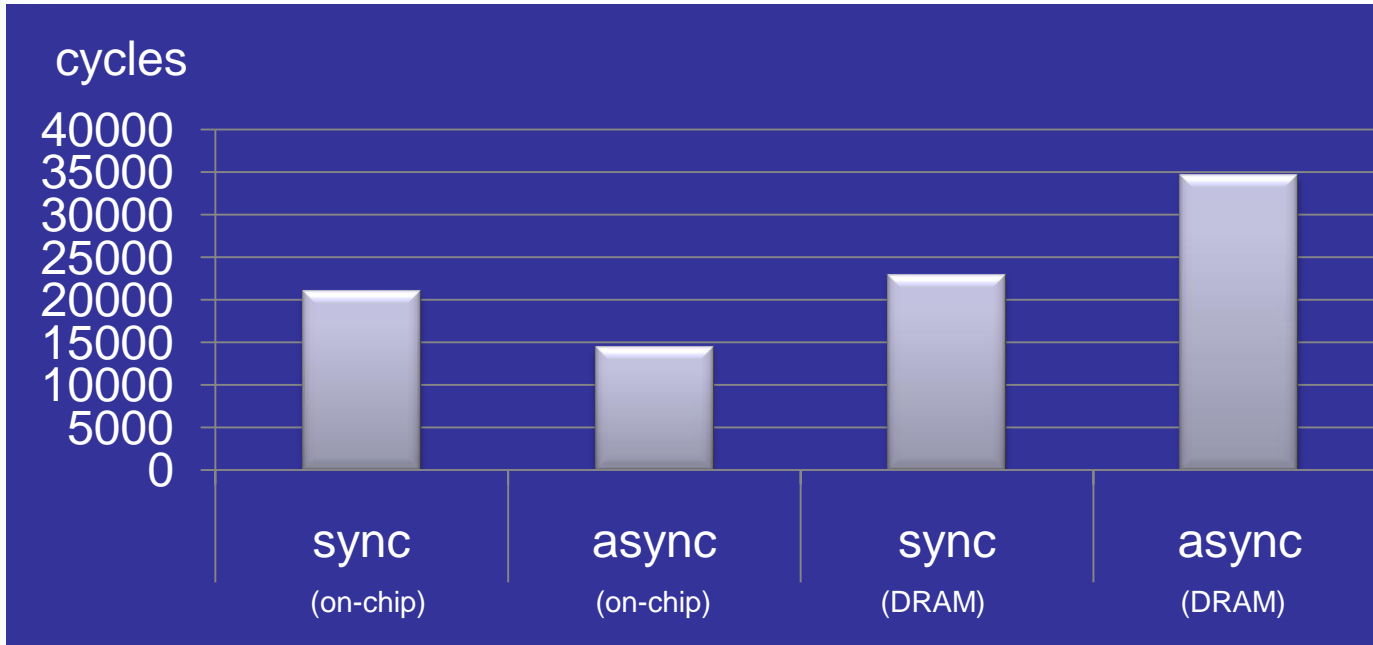


Code run from DDR

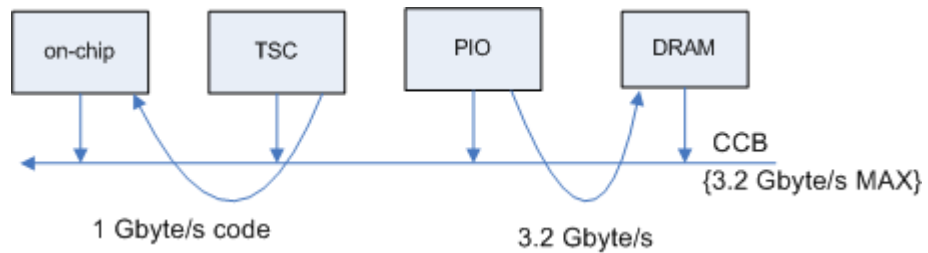
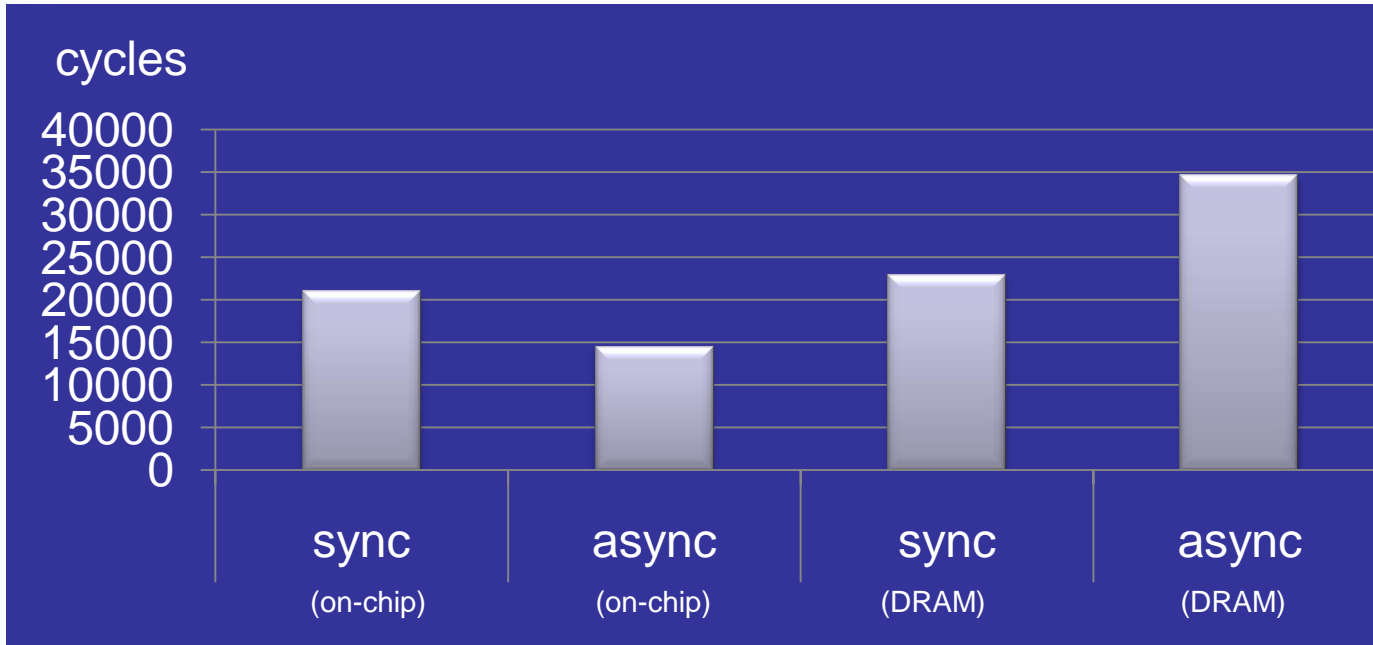


Code run from on-chip

Motivation: Memory bandwidth contention



Motivation: Memory bandwidth contention



Motivation: Limited on-chip memory

- On-chip memories are limited in size, often ranging from 1k – 200k.
- On-chip memory can offer low cycle access and low power.
- Program's footprint is often greater than size of on-chip memory and thus needs to be partitioned.
- Main motivation in previous work on on-chip (scratch pad) memory usage.

Overlays

- Group “related” sub-routines together.
- Dynamic loading and unloading of overlays.
- Works well for cases with large on-chip memories, e.g. both ClearSpeed’s CSX and IBM’s Cell have memories greater than 100K.
- Simply design and implementation.

```
#pragma overlay (perform_compute)
void perform_compute(double * poly p2m,
                    poly double * res)
{ ... }

...
perform_compute(p2m_data_ptr,
                &pe_dbl_array[0]);
...
```

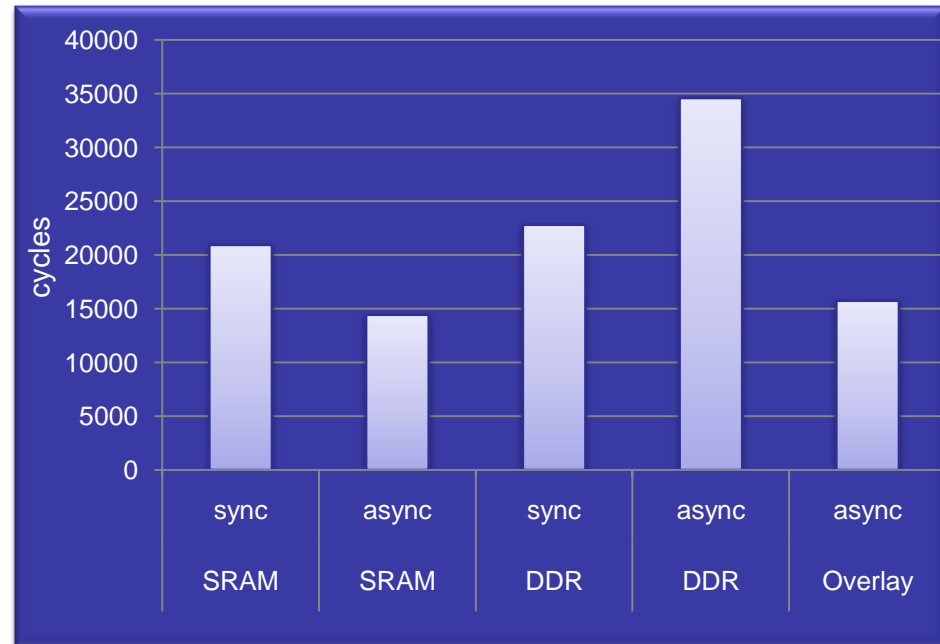
Overlays

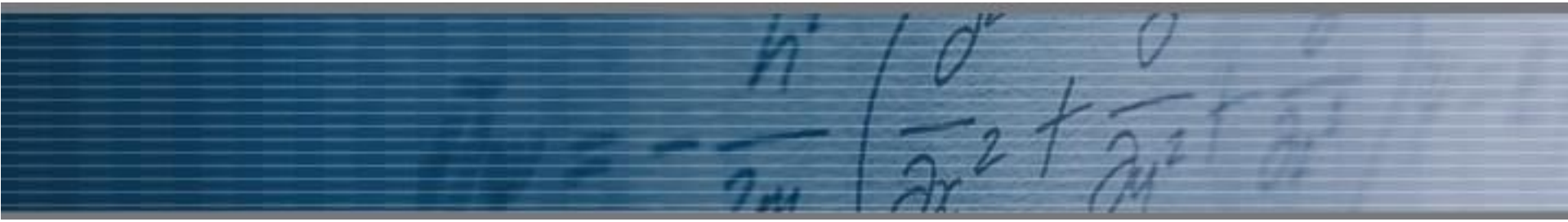
- Group “related” sub-routines together.
- Dynamic loading and unloading of overlays.
- Works well for cases with large on-chip memories, e.g. both ClearSpeed’s CSX and IBM’s Cell have memories greater than 100K.
- Simply design and implementation.

```
#pragma overlay (perform_compute)
void perform_compute(double * poly p2m,
                    poly double * res)
{ ... }

...
perform_compute(p2m_data_ptr,
                &pe_dbl_array[0]);
...

```





A photograph of a chalkboard showing the time-independent Schrödinger equation. The equation is written in white chalk on a dark blue background. It is:
$$\hat{H}\psi = -\frac{\hbar^2}{2m} \left(\frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2} + \frac{\partial^2}{\partial z^2} \right) \psi = E\psi$$

Definition

Overlay sub-routine grouping directive

```
#pragma overlay [clause [, clause] ...]
```

where *clause* can be any of the following:

```
(f1, ..., fn)  
[load(hot)]  
[name(string)]  
[lock]  
[chip(n, ..., m)]
```

with $n \geq 1$

arguments enclosed in [] are optional

Overlay sub-routine grouping directive

(f₁, ..., f_n)

Defines a chunk consisting of the functions f₁...f_n. Each function f_i must exist in the current compilation unit.

load(hot)

Indicates that a particular overlay should be loaded directly into on-chip on program startup.

name(string)

Provides a symbolic name for an overlay. (Intended in to be used for debugging and when grouping overlays together.)

lock

Indicates that a particular overlay should loaded directly into on-chip on program startup and locked for the duration of the program.

chip(n,...,m)

Indicates set of processors, e.g. CSXs or SPUs, that overlay is to be loaded.

A simple example

```
#pragma overlay (a,b) lock
int a(int x) {...}

int b(int x, int y) {... a(x) ... (x+y) ...}

#pragma overlay (c,d) load(hot)
int c(int x) {... b(x,y) ...}

int d(int x, int y) {... for(...) {... c(x,y) ...} ...}

int e(int x) {... a(x) ... b(x,x) ...}

#pragma overlay (e,f,g)
int f(int x) {... e(x) ...}

int g(int x, int y) {... for(...) {... f(x) ...} ... f(x)
...}

int h(int x) {...}

int main(void) {
    ...
    d();
    h();
    g();
    h();
    ...
}
```

A simple example

```

#pragma overlay (a,b) lock
int a(int x) {...}

int b(int x, int y) {... a(x) ... (x+y) ...}

#pragma overlay (c,d) load(hot)
int c(int x) {... b(x,y) ...}

int d(int x, int y) {... for(...) {... c(x,y) ...} ...}

int e(int x) {... a(x) ... b(x,x) ...}

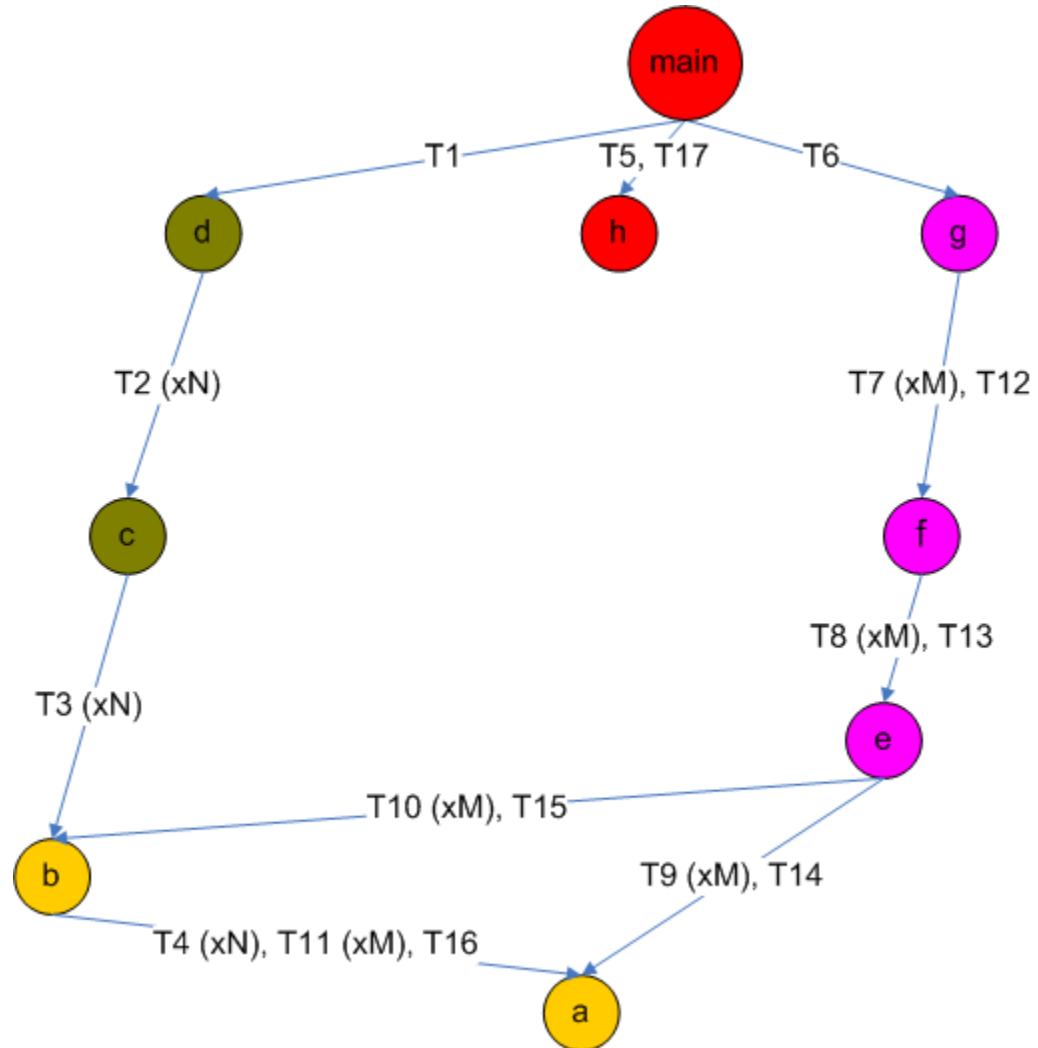
#pragma overlay (e,f,g)
int f(int x) {... e(x) ...}

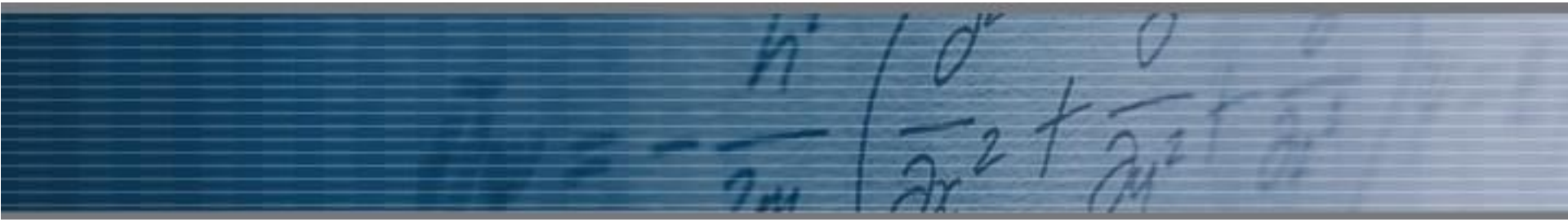
int g(int x, int y) {... for(...) {... f(x) ...} ... f(x) ...}

int h(int x) {...}

int main(void) {
    ...
    d();
    h();
    g();
    h();
    ...
}

```





Implementation

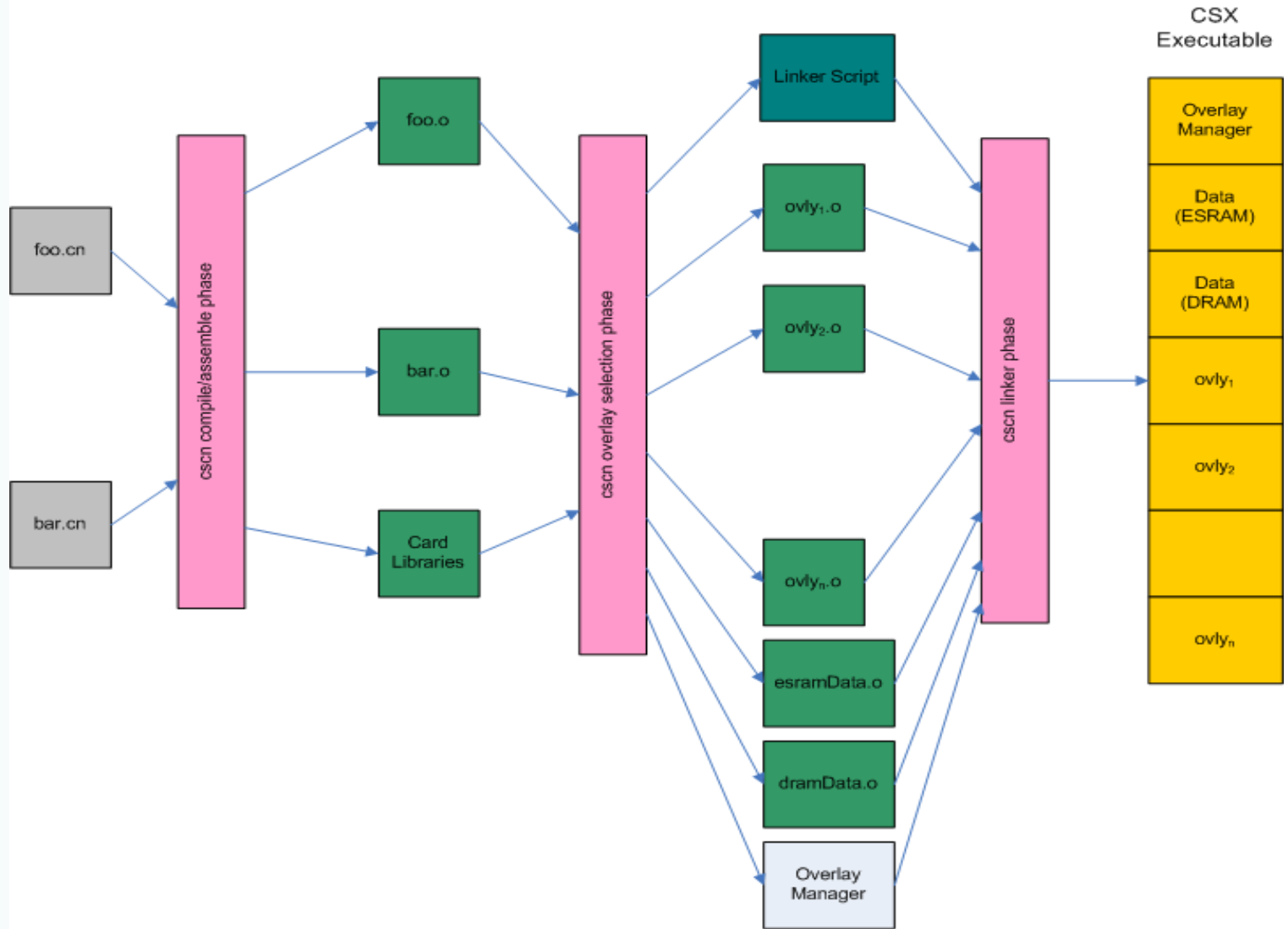
Implementation

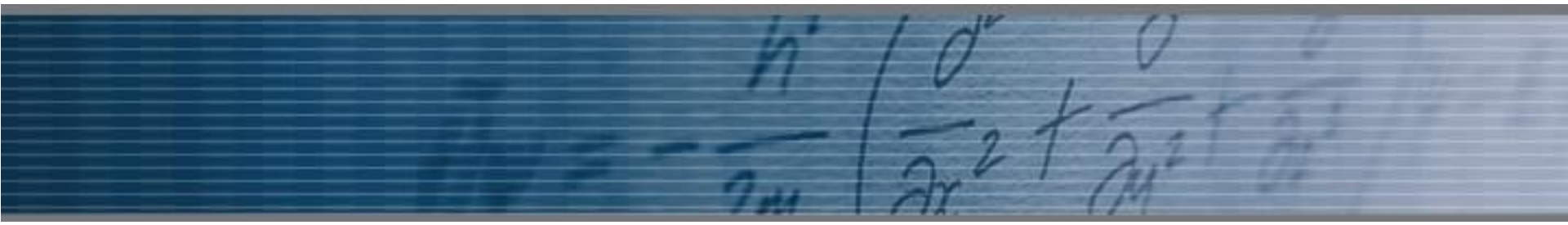
- The compiler (assembler programmer) uses relative branches to jump within a function.
- Intra-function branching transfers control to the runtime which either:
 1. Loads overlay into on-chip memory, patching code so branching into the runtime again is avoided (if called again before eviction) and transfers control to loaded function. (See chaining below.)
 2. Overlay already in on-chip memory, so simply patch address and transfer control. (See chaining below.)
 3. Code is not part of an overlay so simply transfer control to DRAM.
- *Chaining* is an established technique that cuts out unnecessary jumps to the runtime system by modifying the code in the cache.
- Two eviction strategies are supported: *FIFO* and *purge*.

Implementation: function pointers

- These are handled by a runtime mapping from DRAM addresses into on-chip memory using height balanced trees.
- A function pointer dereference causes control to be transferred to the runtime where the DRAM address is mapped to its actual location. (Note, as with other function calls this may mean loading an overlay.)
- As it is possible in C for a function pointer to point to anywhere in memory, it is not enough to simply map to the start of a function in DRAM. Instead an 'address + offset' mapping is necessary.
- In many cases pointer-analysis of the program will allow a statically known hash-table to be used and thus reduce the address mapping overhead.

Compile and link phases

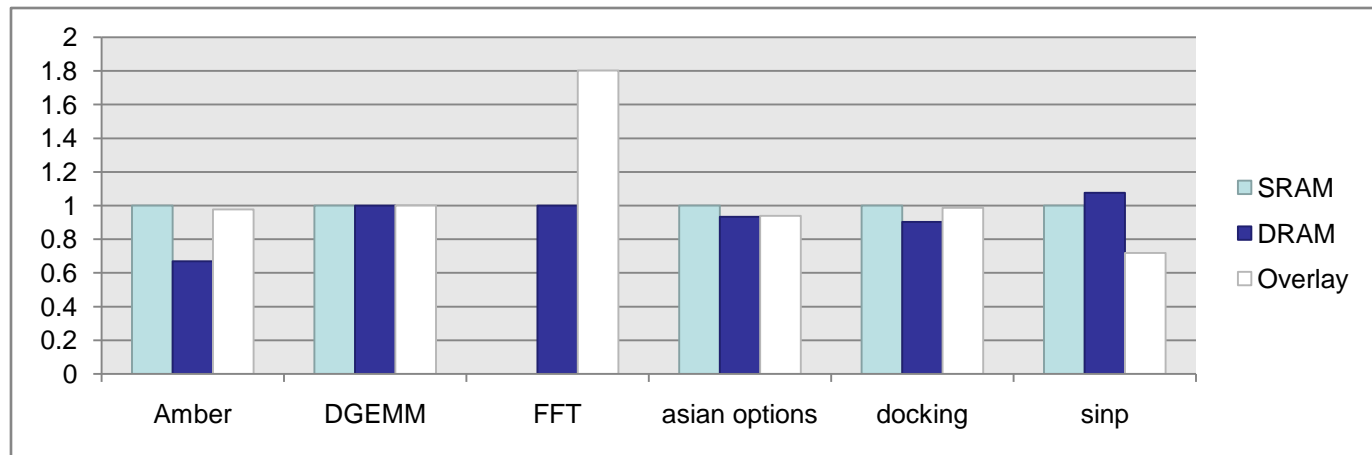




Evaluation

Performance evaluation

Benchmark	Description
Amber	Benchmark for a simulation run calculating molecular mechanical force fields for a CSX optimised version of the Amber 9 Sander Implicit solver.
DGEMM	Single run of benchmark for the BLAS 3 (double precision) matrix multiple.
FFT	Basic benchmark test for ClearSpeed's single precision 2D Fast Fourier transform.
Asian Options	Benchmark for Asian options pricing using a non-closed form Monte Carlo method.
Docking	Benchmark for a simulation run of Bristol University's protein docking system.
sinp	Simple un-optimised benchmark to compute the parallel sin of 96 angles.



Conclusion

- Described a simple approach to dynamic code placement
 - Use driven directive approach.
 - Straightforward to provide toolchain support. (In particular avoids integer programming of other more advanced approaches.)
 - Requires “smart” runtime, using a number of well developed techniques.
- Deployed in production environment.
- Evaluation shows performance on par with running code from on-chip memory for many applications, and in the case when code base is too large, there is a significant increase.

Related work

Overlays

R. J. Pankhurst. Operating systems: Program overlay techniques. *Commun. ACM*, 11(2):119–125, 1968.

A. E. Eichenberger, J. K. O'Brien, K. M. O'Brien, P. Wu, T. Chen, P. H. Oden, D. A. Prener, J. C. Shepherd, B. So, Z. Sura, A. Wang, T. Zhang, P. Zhao, M. K. Gschwind, R. Archambault, Y. Gao, and R. Koo. Using advanced compiler technology to exploit the performance of the Cell Broadband Engine architecture. *IBM Syst. J.*, 45(1):59–84, 2006.

Scratch-pad memories

Federico Angiolini, Francesco Menichelli, Alberto Ferrero, Luca Benini, and Mauro Olivieri. A post-compiler approach to scratchpad mapping of code. In *CASES '04: Proceedings of the 2004 international conference on Compilers, architecture, and synthesis for embedded systems*, pages 259–267, New York, NY, USA, 2004. ACM Press.

Bernhard Egger, Chihun Kim, Choonki Jang, Yoonsung Nam, Jaejin Lee, and Sang Lyul Min. A dynamic code placement technique for scratchpad memory using postpass optimization. In *CASES '06: Proceedings of the 2006 international conference on Compilers, architecture and synthesis for embedded systems*, pages 223–233, New York, NY, USA, 2006. ACM Press.

Jason E. Miller and Anant Agarwal. Software-based instruction caching for embedded processors. In *ASPLOS-XII: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, pages 293–302, New York, NY, USA, 2006. ACM Press.

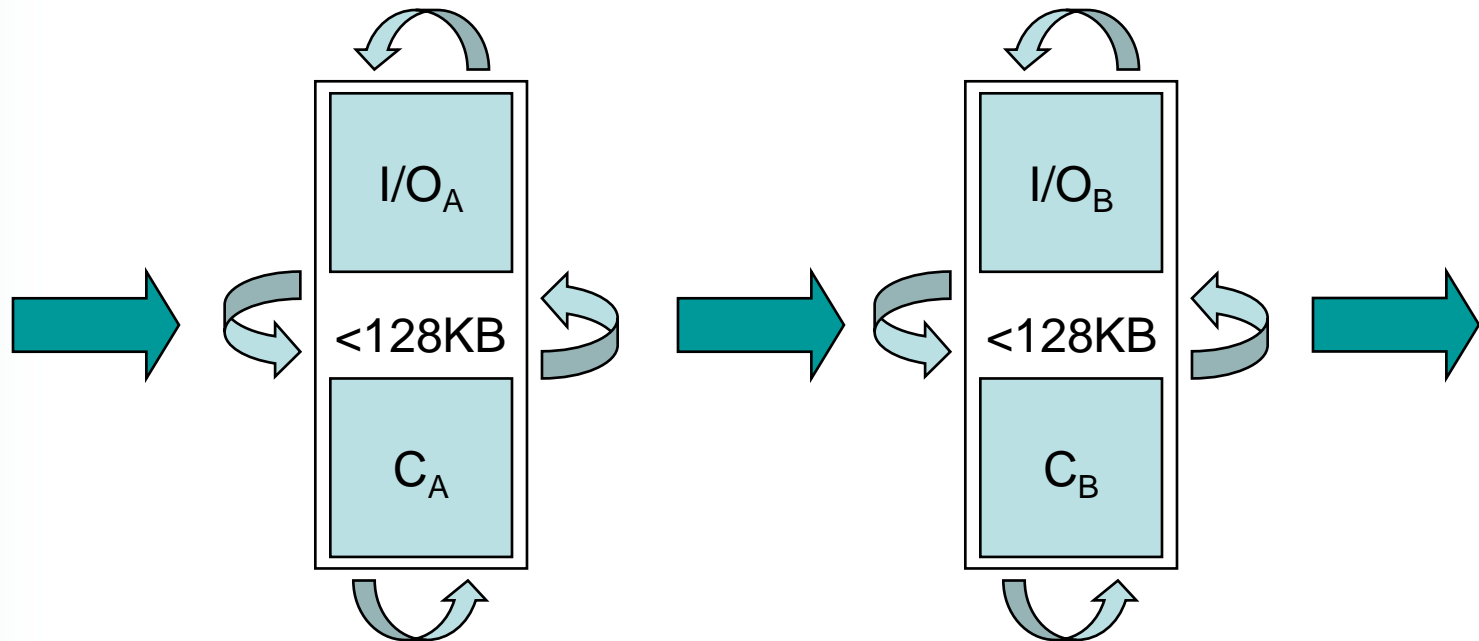
Preeti Ranjan Panda, Nikil D. Dutt, and Alexandru Nicolau. Efficient utilization of scratch-pad memory in embedded processor applications. In *EDTC '97: Proceedings of the 1997 European conference on Design and Test*, page 7, Washington, DC, USA, 1997. IEEE Computer Society.

Sumesh Udayakumaran, Angel Dominguez, and Rajeev Barua. Dynamic allocation for scratch-pad memory using compile-time decisions. *Trans. on Embedded Computing Sys.*, 5(2):472–511, 2006.

Implementation: Using overlays in assembler

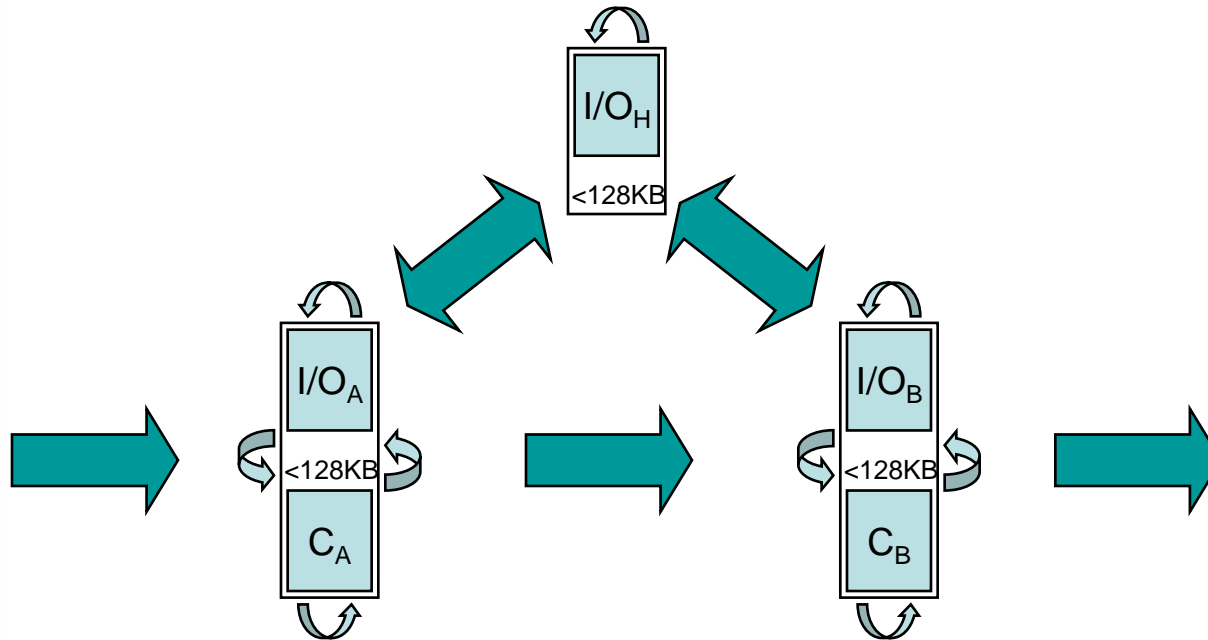
- It is not the intention to provide special syntax for using overlays in assembler.
- The assembler programmer will have to allocate overlays manually and place direct calls into the runtime to transfer control at the desired locations. Not doing so will cause control to transfer to the default location for a particular function, i.e. DRAM unless a function is 'locked' in on-chip memory with a particular overlay.
- Any function written in assembler that is intended for inclusion in an overlay must use relative branches internally.

One function followed by another



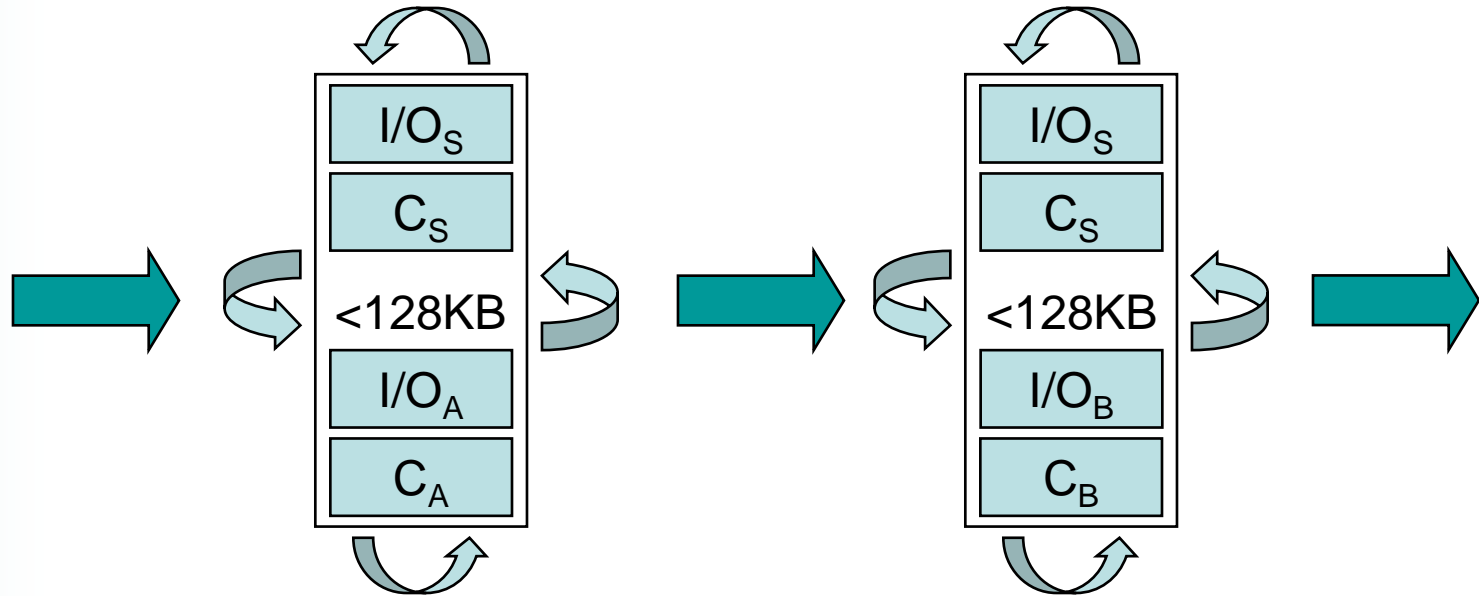
- A sequence of (potentially) large functions, each including its own I/O and Compute (e.g. I/O_A and C_A).
- Assumes all the code for each function fits in on-chip.
- No code needs to stay resident between functions.
- No additional DRAM activity during function transitions from A→B.

A series of functions with continuous host I/O



- A sequence of (potentially) large functions, each including its own I/O and Compute (e.g. I/O_A and C_A).
- Assumes all the code for each function fits in on-chip.
- The host I/O (I/O_H) may be running continuously, transferring data for subsequent functions via DMA.
- The total code size of I/O_H plus all the code for another function may exceed the size of the on-chip.
- (In addition, I/O_H may be hammering the DRAM via DMA while a function transition from $A \rightarrow B$ occurs).

A series of functions with some shared code



- A sequence of (potentially) large functions, each including its own I/O and Compute (e.g. I/O_A and C_A).
- Includes a set of code that stays resident across the sequence I/O_S and C_S (e.g. runtime functions such as memcpym2p et al).
- Assumes all the code for each function fits in on-chip.