Decoupled Access-Execute on ARM big.LITTLE

Anton Weber
Abstract

Decoupled Access-Execute on ARM big.LITTLE

Anton Weber

Decoupled Access-Execute (DAE) presents a novel approach to improve power efficiency with a combination of compile-time transformations and Dynamic Voltage Frequency Scaling (DVFS). DAE splits regions of the program into two distinct phases: a memory-bound access phase and a compute-bound execute phase. DVFS is used to run the phases at different frequencies, thus conserving energy while caching data from main memory and performing computations at maximum performance.

This project analyses the power-savings and performance impact of DAE on the ARM architecture for the first time, a platform that is prominent in the mobile market where battery life is of particular significance. We target ARM big.LITTLE specifically, a heterogeneous hardware platform where code can not only be executed at different frequencies, but also transparently on different microarchitectures with individual energy and performance characteristics. As a result, this architecture enables hardware support for new DAE concepts that have not been previously available.

As a first step towards new DAE compiler passes, we demonstrate the methodology for a thread-based DAE implementation that specifically targets the hardware features of ARM big.LITTLE. Finally, we manually apply it to a selection of benchmarks and analyse how this DAE implementation performs on the Samsung Exynos 5422 big.LITTLE SoC and identify the strengths and limitations of the implementation on current hardware.

Handledare: Kim-Anh Tran
Ämnesgranskare: Alexandra Jimborean
Examinator: Edith Ngai
IT 16 059
Tryckt av: Reprocentralen ITC
1 Introduction

Designed for embedded devices with strict size and energy constraints [1], ARM was quickly adopted in modern mobile hardware, such as smartphones and tablets, making it the de-facto standard in these devices today. One of the main reasons for this is the low-power RISC design, allowing for small, energy efficient chips that at the same time are powerful enough to run modern mobile operating systems.

While chip designs have been able to keep up with the rapid development of the mobile segment in the past, ARM, just like its competitors, is facing new challenges with the increasing demand on performance and battery life in portable devices and the end of Dennard scaling where it is no longer possible to lower transistor size and keep the same power density. To address this issue, ARM developed a new heterogeneous design named big.LITTLE that aims at combining different CPU designs on the same system on chip (SoC). It allows to use more energy efficient cores to conserve power during simple tasks and idle times and switch over to faster, more power hungry cores when performance is needed.

An alternative approach to this issue is to develop new software techniques that allow for the existing hardware to be used more efficiently. Decoupled Access-Execute [2] has been developed as a way to improve performance and energy efficiency on modern CPUs through static transformations at compile-time. It groups code into two coarse-grained phases where the access phase contains memory-bound instructions and all compute-heavy code is grouped in the execute phase. Using Dynamic Voltage Frequency Scaling (DVFS) these phases can be executed at different CPU frequencies and voltages. During the memory-bound phase the CPU is stalled waiting for data to arrive from memory and can be clocked down to conserve energy without performance loss. Once the code reaches the execute phase, the CPU can be scaled back to perform the compute-heavy tasks at maximum performance. As a result, processors can save energy by executing parts of a program at lower voltage or frequency without slowing down the overall execution. This work has been our starting point.

With this project we try to bring the two concepts together, not only to demonstrate the advantages of Decoupled Access-Execute on ARM, but also to adapt the previous ideas to the new architecture and take advantage of the unique features that the heterogeneous design of ARM big.LITTLE has to offer. In particular, we want to benefit from two different core designs with individual performance and energy characteristics being available to run decoupled execution. We develop a new transformation pattern and create prototypes for a selection of benchmarks as a proof-of-concept with the goal to create compiler passes that can apply the techniques to any suitable input program in the near future. While doing so, we want to follow the DAE philosophy of reducing energy consumption without sacrificing performance.

2 Background

2.1 Decoupled Access-Execute

While caches and hardware prefetchers have been introduced to decrease latency when accessing data from main memory, it still remains a common bottleneck in current computer architectures. With the processor waiting for data to arrive, there is not only a negative impact on the overall runtime of programs, but also on energy consumption.

One reason for this is that the processor remains at full frequency while stalled in order to compute at full performance as soon as the data arrives. Lowering the frequency to reduce energy consumption during stalls would result in slower computations and
further increase in program execution time. As an ideal approach, the memory-bound instructions that stall the CPU should be executed at low frequencies while the processor should run at maximum frequency for compute-bound parts of the program.

Spiliopoulos et al. [3] explored this possibility and created a framework that detects the memory- and compute-bound regions in the program and scales the CPU’s frequency accordingly. Their work shows that this is indeed a viable approach but only if the regions are coarse enough.

One of the reasons for this is that using current techniques, such as DVFS, switching the frequency or core voltage on the processor involves a significant latency, preventing us from just switching between low and high frequencies as rapidly, as the ideal approach would require.

With Decoupled Access-Execute (DAE), Koukos et al. [4] proposed a solution to this problem by grouping memory- and compute-bound instructions in a program, creating larger code regions and thus reducing the number of frequency switches required. The result are two distinct phases, referred to as access (memory-bound) and execute phase (compute-bound).

Jimborean et al. [2] created a set of compiler passes in LLVM allowing for these transformations of the program to be performed statically at compile-time. These passes create the access phase by duplicating the original code and stripping the copy of all instructions that are not memory reads or address computations. The original code becomes the execute phase.

While the added access phase introduces overhead initially, the execute phase will no longer be stalled by memory accesses, as the data will be available in the cache. In addition, the access phase allows for more memory level parallelism, potentially speeding up main memory accesses compared to the original code.

```c
for(;;) {
    for(granularity) {
        // select Access
        loop body
    }
}
```

Figure 1: Loop transformation with multiple access phases in SMVDAE. Figure by Koukos et al. [5]

Software Multiversioned Decoupled Access-Execute (SMVDAE) [5] deals with the difficulties of transforming general purpose applications. Here complex control flow and pointer aliasing make it hard to statically reason about the efficiency of the transformations. The SMVDAE compiler passes apply the transformations with a variety of parameters that allow to find the best performing options for the given code dynamically. With this approach, Koukos et al. were able to achieve on average over 20% energy-delay-product (EDP) improvements across 14 different benchmarks.

To create coarse grained code regions, DAE targets large loops within the programs.
These loops are split into smaller slices that typically consist of several iterations of the original loop. Figure 1 illustrates how this loop chunking transforms the original code in SMVDAE. The goal of this is to prefetch as much data as possible into the cache for the execute phase. As the amount of data accessed in each loop iteration depends on the task, the optimal size for the slices, the granularity, is benchmark-and-cache dependent.

2.2 ARM big.LITTLE

Since its first prototype, the ARM1 from 1985, many iterations of ARM CPU cores have been developed. One key characteristic that distinguishes all these core designs from competitors like Intel and AMD is the RISC architecture. Instead of implementing a set of complex instructions in hardware, the RISC architecture shifts the complexity away from the chip and towards the software. This makes RISC compiler tools more complex but also allow for much simpler hardware designs on the CPU. As a result, ARM has been able to create small, energy efficient chips that have seen great popularity in the embedded and mobile segment where size and battery life are major factors.

A recent development by ARM is the big.LITTLE architecture. First released in 2011, this heterogeneous design combines fast, powerful cores with slower, more energy efficient ones on the same SoC, allowing the devices to conserve power during small tasks and idle states but at the same time deliver high performance when needed. While the big and LITTLE processor cores are fully compatible from an instruction set architecture (ISA) level, they feature different microarchitectures. big cores are characterized by complex, out-of-order designs with many pipeline stages while the LITTLE cores are more simple, in-order processors. In comparison, the ARM Cortex-A15 (big) pipeline has 15 integer and 17-25 floating point pipeline stages, while the Cortex-A7 (LITTLE) only has 8 pipeline stages.

In modern ARM SoCs, CPU cores are grouped in clusters. Each core has an individual L1 data and instruction cache but shares the L2 cache with other cores in the same cluster. CPU clusters and other components on the chip, such as GPU and peripherals, are connected through a shared bus. ARM provides reference designs for these interconnects, but manufacturers commonly also use custom implementations in their products. The interconnect uses ARM’s Advanced Microcontroller Bus Architecture (AMBA) and provides system-wide coherency through the AMBA AXI Coherency Extensions (ACE) and AMBA AXI Coherency Extensions Lite (ACE-Lite). These protocols allow memory coherency across CPUs, one of the prerequisites for big.LITTLE processing.

ARM big.LITTLE usually features two clusters: one for all big cores and one cluster containing all LITTLE cores. These designs allow three different techniques for runtime migrations of tasks: cluster switching, CPU migration and Global Task Scheduling (GTS).

Cluster switching was the first and most simple implementation. Only one of the two clusters is active at a time while the other is powered down. If a switch is triggered, the other cluster is powered up, all tasks are migrated and the previously used cluster is deactivated until the next switch. In CPU migration, each individual big core is paired with a LITTLE core. Each pair is visible as one virtual core and the system can transparently move a task between the two physical cores without affecting any of the other pairs. GTS is the most flexible method, as all cores are visible to the system and can all be active at the same time.

Cluster switching and GTS also allow for asymmetric designs, where the number of big and LITTLE cores does not necessarily have to be equal.
3 Related Work

3.1 Inter-core prefetching

Kamruzzaman, Swanson and Tullsen [9] investigated how a multi-core system can be used to improve performance in single-threaded applications. Their inter-core prefetching technique uses helper threads on different cores to prefetch data. The original compute thread is migrated between these cores at specific points in the execution to benefit from the data that has been brought into the caches. This technique performs transformations in software without the need of special hardware and, similar to the DAE approach, targets large loops within a program.

With memory-bound applications, they have been able to achieve up to 2.8x speedup and an average energy saving between 11 and 26%. It also shows the advantages of moving the prefetches to a different core in comparison to previous simultaneous multithreading (SMT) approaches. This prevents from competing for CPU resources with the main thread and avoids the negative impact on L1 cache behaviour. On the other hand, it is also mentioned that this approach has downsides, such as problems with cache coherence when working on the same data in the main and helper threads.

This concept has similarities to the methodology we design for DAE on big.LITTLE as described further below, where we are utilizing multiple cores simultaneously with one core dedicated to provide the main execution with data through prefetches. Thus the work of Kamruzzaman, Swanson and Tullsen will help us to analyse and evaluate our solution.

3.2 big.LITTLE task migration

big.LITTLE implementations currently choose between two distinct methods for task migration. The first method relies on CPU frequency frameworks, such as cpufreq, and works with cluster switching and CPU migration. When a certain performance threshold is reached, tasks are migrated to another core or a cluster switch is triggered. This is comparable to traditional DVFS techniques with the difference that lower power is not represented by a change in CPU voltage or frequency but by a migration to a different core or cluster [7].

Global Task Scheduling relies on the OS scheduler to migrate tasks. In this model the scheduler is aware of the different characteristics of the cores in the system and creates and migrates tasks accordingly. For this, it tracks the performance requirements of
threads and CPU load on the system. This data can then be used together with heuristics to decide the scheduling behaviour \cite{7, 8}. As all cores are visible to the system and can be active at the same time, this method is regarded as the most flexible. Manufacturer white papers show that GTS improves benchmark performance by 20\% at similar power consumption compared to cluster switching on the same hardware \cite{8}.

3.3 Scheduling on heterogeneous architectures

Chen and John \cite{10} analysed scheduling techniques on heterogeneous architectures. For this they created a model that bases scheduling decisions on matching the characteristics of the different hardware to the resource requirements of the tasks to be scheduled. In their work, they consider instruction-level parallelism (ILP), branch predictability, and data locality of a task and the hardware properties hardware issue width, branch predictor size and cache size. Using this method they reduce EDP by an average of 24.5\% and improve throughput and energy savings. While DAE does not consider the same workload characteristics, we will be taking a related approach with this work by matching the different phases to the appropriate core type within the heterogeneous big.LITTLE design.

Van Craeynest et al. \cite{11} created a method to predict how the different cores in single-ISA heterogeneous architectures, including ARM big.LITTLE, perform for a given type of task and developed a dynamic scheduling mechanism based on their findings. One aspect that is mentioned in their work is that the simple, in-order cores perform best on tasks with high ILP while the complex out-of-order cores benefit from memory-level parallelism (MLP) or where ILP can be extracted dynamically. As a conclusion, choosing cores based on whether the execution is memory- or compute-intensive without taking those aspects into account can lead to suboptimal performance.

Decoupled execution currently purely focuses on the memory- or compute-bound properties to divide the tasks into the different phases. While the work of Van Craeynest et. al. \cite{11} evaluates benchmarks as a whole and DAE is applied on a much finer scale within individual functions of the program, their insights are important to take into account when moving DAE to a heterogeneous architecture. When deciding which tasks to consider for the individual phases for example, taking the level of ILP and MLP into account becomes important as the phases can now be scheduled on different types of cores.

4 Methodology

Our methodology applies DAE on ARM and takes advantage of the heterogeneous hardware features on the big.LITTLE architecture. With two different types of processors available on the same system, energy savings can now be a result of lower core frequencies and running code on the simpler, more energy efficient microarchitecture of the LITTLE cores. A straight-forward way to benefit from this in decoupled execution, is to place our access and execute phase onto the different cores.

Running the two phases on several CPUs also eliminates DVFS latency, as the cores can constantly be kept at ideal frequencies throughout the entire execution. Since the cores are located on different clusters, we are no longer prefetching the data into a shared cache and are instead providing the execute phase with prefetched data through coherence.

We have chosen to implement this approach for a selection of benchmarks.
4.1 Transformation pattern

As our future goal is to have a set of compiler passes that can transform any input program, we design our new DAE transformations as a step by step process. The following sections describe this pattern and illustrate it on a sample program in pseudocode. The sample program in figure 4 is used as a running example.

Although some of the steps have remained largely unchanged from current DAE compiler passes, we have chosen to implement this approach as prototypes in C code first. This allows us to adjust parameters and details in the implementation more rapidly and with more flexibility.

```c
// Main thread
void do_work() {
    for(i=0; i<N; i++){
        c[i] = a[i+1]+b[i+2]
    }
}
```

Figure 4: Running example: Original program

**Spawning threads for access and execute phase**

Similar to the current DAE compiler passes, we are targeting large loops within the benchmarks. The loop is duplicated, but instead of executing the access and execute phases within the same thread, we move them to individual cores (see figure 3). This is done by creating two threads: one for the access- and one for the execute phase.
The threads are created using the Linux POSIX thread interfaces (Pthreads). Configuring the attributes to the Pthread calls enables us to manually define CPU affinity and spawn the execute phase on a big core and the access phase on a LITTLE core. This allows us to benefit from the flexibility of Global Task Scheduling and as our phases are meant to remain on the same core for the entire execution, we can avoid task migration and any of the associated overhead entirely.

For the first transformation step this effectively means copying the loop twice and placing it into an empty function each. These two new functions will become our access and execute phase. The original loop is replaced with the calls required to spawn two threads, one for each of the new functions, and join them when they finish. Figure 5 shows the output of applying this transformation to the example program.

```c
//Main thread
void do_work() {
    spawn(access_thread, access)
    spawn(execute_thread, execute)
    join(access_thread)
    join(execute_thread)
}

//Access thread
void access() {
    for(i=0;i<N;i++){
        c[i] = a[i+1]+b[i+2]
    }
}

//Execute thread
void execute() {
    for(i=0;i<N;i++){
        c[i] = a[i+1]+b[i+2]
    }
}
```

Figure 5: Running example: Duplicating the loop and executing the copies as individual threads

**Generating access and execute phase**

Similar to previous DAE approaches, our method includes control flow, loads and memory address calculations in the access phase. Stores are buffered and unlikely to stall the processor and are hence not included in this phase [4]. Once all irrelevant instructions have been removed from the access phase, we can potentially optimize the resulting code during the compilation step, such as removing dead code or control flow that is no longer needed as a result of our changes.

The execute phase remains unchanged after the loop chunking as we can issue the same data requests to the interconnect as the original code. All accesses to the data that is prefetched in the access phase automatically benefit from it being available closer in the memory hierarchy, as the request will be services by the other cluster transparently.
Both phases contain address calculations as they are required to both prefetch and load the data for calculations. This means that we are now calculating addresses twice, introducing additional instructions compared to the original program. As in previous DAE implementations, we aim at compensating for this overhead through the positive side-effects of decoupled execution.

In our pattern, this step involves chunking the loop (i.e. creating an inner and outer loop) and replacing instructions in the access phase as described above (see figure [6]).

```
// Main thread
void do_work() {
    spawn(access_thread, access)
    spawn(execute_thread, execute)
    join(access_thread)
    join(execute_thread)
}

// Access thread
void access() {
    // Outer loop
    offset=0
    for(j=0; j<(N/granularity); j++){
        // Inner loop
        for(k=0; k<granularity; k++){
            i=offset+k
            prefetch(a[i+1])
            prefetch(b[i+2])
        }
        offset+=granularity
    }
}

// Execute thread
void execute() {
    // Outer loop
    offset=0
    for(j=0; j<(N/granularity); j++){
        // Inner loop
        for(k=0; k<granularity; k++){
            i=offset+k
            c[i]=a[i+1]+b[i+2]
        }
        offset+=granularity
    }
}
```

Figure 6: Running example: Chunking the loop and removing instructions from the access phase

Synchronization

As we are parallelizing decoupled execution, synchronization is required to enforce two rules: First, the execute phase must not start computing before prefetching has finished, as it can only benefit from decoupled execution when the data is available
in the cache as it is requested. And second, the access phase can not start the next slice before the execute phase has completed the current. As cache space is limited, prefetching the next set of data will potentially evict previously prefetched cache lines. As seen in figure 7, running access and execute phases in turn on the two cores can be achieved using a combination of two locks.

Figure 7: Synchronization between individual phases

Figure 8 shows how we can implement such a locking scheme in this transformation step. Each phase waits on one of these locks before starting the next chunk.

**Access lock (blue):** The first lock is used to start the access phase and unlocked at the end of each slice in the execute phase. It is initially unlocked, so that the access phase can start as soon as the thread spawns.

**Execute lock (red):** The second lock signals the execute phase to continue once the access phase finished the current slice. The execute phase follows the same pattern with the difference that its first lock is initialized as locked, making sure that the phase does not start before the first access slice has been completed.
Figure 8: Running example: Adding synchronization between access and execute phase

4.2 Optimizations

The pattern above describes a basic implementation of DAE on big.LITTLE. Further optimizations to this methodology can improve results significantly in some scenarios. While only the first optimization has been used as part of this work, all of the methods
Reducing thread overhead

As we create a pair of threads every time a loop is executed, the overhead of setting up, spawning and joining the threads can degrade performance noticeably when the program executes the loop frequently. A solution to this is to keep the threads running over the course of the program and provide them with new data on every loop execution. This thread-pool approach does not come without a downside as the threads need to be signalled when new data is available and the next loop should be executed. In many cases the benefits outweigh the added overhead and we decide experimentally whether to apply this optimization to the chosen benchmarks.

Overlap

Running the access and execute phases in parallel introduces a new way to reduce overall execution time by overlapping the two phases for each individual slice. While it is problematic to start the execute phase too early, as the data has not yet arrived in the cache of the LITTLE cluster, it does not have to wait for the entire slice to be prefetched.

In other words, to benefit from prefetched data in the execute phase, the prefetch has to be completed by the time a particular data set is needed. Hence, any prefetches for data that is required at a later stage can still be pending when the execute phase starts as long as they complete by the time the data is requested.

Figure 9 illustrates one possible scenario where the execute phase is started before the last data set has been prefetched. As the data set is only required towards the end of the current execute slice, the access phase can use the early stages of the execute thread to prefetch the last set of data in parallel.

Achieving the correct timing for this can be difficult, as the two phases are executed at different speeds and the execute phase processes the data at a different rate than it is prefetched in the access phase.

![Figure 9: Overlapping access and execute phase](image)

In fact, as we are using the Preload Data (PLD) instruction [12] to prefetch data in the access phase, we are already dealing with overlap in the current implementation to some degree. This prefetch hint is non-blocking, meaning that it will not wait for the data to actually arrive in the cache before completing. Hence, finishing the access
phase only guarantees that we have issued all hints while any number of them might not have prefetched the data into the cache yet.

4.2.1 Timing-based implementation

As we are expecting the synchronization overhead to be our main problem with the methodology described above, reducing or removing it entirely would greatly improve how the implementation performs.

While we generally need synchronization when dealing with multiple threads that work together, we have the advantage that we do not affect the correctness of the program by starting the execute phase early or late. Incorrect timing merely affects the performance of the execute phase, as the operations within the access phase are limited to address calculations and prefetches - i.e., operations that have no side-effects. As a matter of fact, we already have a loose synchronization model as a result of the non-blocking prefetches described in the section above.

![Figure 10: Timing-based DAE](image)

This allows us to go with a more speculative approach. Instead of locking, we can suspend the two threads and let the access and execute phases wait for a specific amount of time before processing the next slice. The amount of wait time should correspond to the execution time of the other phase. Runtime measurements of the individual phases provide us with a starting point for these sleep times, while the exact values that perform best can be found experimentally. As a result, we can approximate a similar timing between the two threads without any of the locking overhead (see figure [10]).

The main problem with this alternative is that it is difficult to reliably approximate the timing of the two threads as our system does not guarantee real-time constraints. Other programs running on the same system and the operating system itself can interfere with the DAE execution and affect the runtime of individual slices. Suspending the threads can be inaccurate, too, as using functions like `nanosleep` can be inaccurate and resume threads too late or return early.

The consequence in these cases is that the two threads can go out of sync, working on two different slices and negating all DAE benefits. The impact and likeliness of this to occur is dependent on many factors, including the number and size of the slices.

A trade-off would be a hybrid solution that only locks periodically to synchronize the two threads to the same slice. A group of slices would still be executed lock-free (i.e., with reduced overhead) and the synchronization point would prevent one of the phases from running too far ahead. In a more advanced implementation, these moments can
also be used to adjust the sleep times of the individual threads dynamically to adapt to the current execution behaviour caused by system load and other external factors.

```c
//Execute thread
void execute() {
    //Outer loop
    offset=0
    for(j=0;j<(N/granularity);j++){
        //Inner loop
        nanosleep(S)
        for(k=0;k<granularity;k++){
            i=offset+k
            c[i]=a[i+1]+b[i+2]
        }
        offset+=granularity
    }
}
```

Figure 11: Replacing both locks with a single sleep call in each phase

5 Evaluation

5.1 Experimental Setup

Test system

We evaluate the benchmarks on an ODROID-XU4 single-board computer, running the Samsung Exynos 5422 SoC. This ARMv7-A chip features one big and one LITTLE cluster. Each cluster shares the L2 cache while each individual core has a private L1 data and private L1 instruction cache available. The 2 GB of LPDDR3 main memory is specified with a bandwidth of 14.9 GB/s. Table 1 contains more details about the processors on this chip.

<table>
<thead>
<tr>
<th></th>
<th>big cluster</th>
<th>LITTLE cluster</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of cores</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>Core type</td>
<td>Cortex-A15</td>
<td>Cortex-A7</td>
</tr>
<tr>
<td>(f_{max})</td>
<td>2 GHz</td>
<td>1.4 GHz</td>
</tr>
<tr>
<td>(f_{min})</td>
<td>200 MHz</td>
<td>200 MHz</td>
</tr>
<tr>
<td>L2 cache</td>
<td>2MB</td>
<td>512 kB</td>
</tr>
<tr>
<td>L1d cache</td>
<td>32kB</td>
<td>32 kB</td>
</tr>
</tbody>
</table>

Table 1: Exynos 5422 specifications

The operating system is a 64 bit Ubuntu Linux with a kernel maintained by ODROID that contains all relevant drivers to run the scheduler in GTS mode. To reduce scheduler interference, two processor cores are removed from the scheduler queues using the `isolcpus` kernel option. The benchmarks are cross-compiled on an Intel-based x86 machine using LLVM 3.8 [13].
Measurement technique

Measurements through the Linux perf.events interface, while convenient to set up, have shown to produce too much overhead - especially with frequent, fine-grained measurements. Instead, we directly access the performance statistics available as part of the on-chip Performance Monitor Units (PMUs). In addition to a dedicated cycle counter, we have a number of configurable counters at our disposal (4 per A7 core and 6 on the Cortex-A15 [14, 15]). One of these event counters on each processor is set to capture the number of retired instructions [12]. The cycle counters are configured to increment once every 64 clock cycles to avoid overflows in the 32 bit registers when measuring around larger code regions. We enable PMU counters and allow user-space access through a custom kernel module. This enables us to read the relevant register values through inline assembly instructions from within our code.

5.2 Evaluation criteria

As the goal of this project is to improve energy efficiency of programs without sacrificing performance, these two aspects will be our main criteria for evaluating the prototypes.

Similar to Software Multiversioned Decoupled Access-Execute, we generate multiple versions with different granularities for each benchmark to find the optimal setting.

Performance: We measure the performance impact with the overall runtime of the benchmark. While previous DAE work has shown to speed up benchmark execution in some cases, we now expect synchronization and coherence overhead to affect our results.

Energy: Without a sophisticated power model, speeding up the code run on the big core, together with the individual execution times for each phase, are our main indicators in terms of energy savings. Speeding up the execute phase means that the big core will be active for less time and hence consume less power. On the other hand, these savings are only relevant if the overall overhead is small enough not to negate this effect.

Measurement considerations: Changing the original benchmark into a threaded version introduces synchronization overhead. This is reflected in the runtime measurements, where faster execution does not only result from data being available faster, but also from lower synchronization overhead. This means that we need different measurements to evaluate how much we speed up the execute phase purely by providing the data from our warmed up cache. Currently, the most accurate way to measure this is to see how many CPU cycles the calculations in the execute phase need to finish. If the data is not available in cache, the instructions will take longer time to execute. On the other hand, if the data can be brought in through coherence, the core will spend less time stalled waiting for it to arrive and finish the instructions in less cycles. The baseline for the comparison is the reference version of each benchmark. While loop chunking alone can affect performance, the impact has shown to be negligible for the benchmarks we evaluate.

A common unit to visualize these results is instructions per cycle (IPC). For this we capture the cycle and instruction count in the execute phase individually and calculate the IPC. The samples are taken around the inner loop of the execute phase (see figure [12] to avoid capturing the execution of the other phase and locking overhead. These are measured separately.
Figure 12: Measuring around the inner loop

Benchmarks: For the evaluation of our approach we have modified two benchmarks from the SPEC 2006 suite, libquantum and LBM, and CIGAR [16], a genetic algorithm search. All three benchmarks have been considered by Koukos et al. [4] in the initial task-based DAE framework and allow us a comparison of the DAE on big.LITTLE to previous results.

Each benchmark has individual characteristics and memory access patterns which have an impact on how the prototypes perform. While libquantum and CIGAR are considered memory-bound, LBM has been classified as intermediate [4].

5.3 Benchmark results

LBM

The loop we target in LBM performs a number of irregular memory accesses with limited control flow. The if-conditions only affect how calculations are performed in the execute phase while the required data remains the same in all cases. This results in a simplified access phase without any control flow, as we can always prefetch the same values, independent of which path in the control flow graph the execute phase takes. Most of the values that are accessed from memory are double-precision floating points.

The irregular access pattern and memory-bound calculations are an ideal target for decoupled execution. And in fact, the benchmark results show that we are able to improve the IPC of the execute phase by up to 31% by prefetching the data on the LITTLE core.

The total benchmark runtime increases significantly as we lower the granularity. A smaller granularity increases the number of total slices and with that also the number of synchronizations performed between the two phases. This additional locking overhead results in a penalty to overall runtime. The loop is executed several times based on an input parameter for the benchmark, which multiplies this negative effect.

Figure 13 relates these two findings, the IPC speed-up and the overall benchmark slow-down, to each other. While we observe the best runtime at large granularities, choosing a slightly higher overall slow-down results in much better execute phase...
performance (i.e. we spend less time on the big core). This is discussed in more detail further below.

Figure 13: Improving the execute phase versus slowing down overall execution in LBM

CIGAR

In this case we apply DAE to a function with a higher degree of indirection in the memory accesses. The calculations themselves are relatively simple and compare two double values within a struct to determine a new maximum and swap two values within an integer array.

The results show that we can achieve slightly better IPC improvements compared to LBM with a peak speed-up of 37%. The slow-down of overall execution time at lower granularities is still significant, yet not as extreme as in the previous benchmark. This can be explained by the smaller loop size (i.e. the same granularity results in less total slices) and the fact that the loop is only executed once as part of the benchmark.

The trend of the IPC graph indicated that we can potentially speed up the execute phase even further by lowering the granularity. Yet, as the overhead at low granularities increases significantly, any improvement in IPC would be negated.

libquantum

While the other two benchmarks spawn threads every time they execute their loop, frequent calls to our target loop in libquantum make it a good candidate for the thread-pool optimization. As we in fact have been able to reduce the overhead noticeably in this case, we have taken all measurements with the thread-pool variant.

The loop itself has a regular access pattern and performs a single bitwise XOR operation on a struct member on each iteration. Despite the memory-bound nature of this loop, we only observe a maximum execute phase IPC speed-up of 6.7% (see figure 15). This is an interesting finding, as previous DAE evaluations of libquantum show improvements in energy and performance [2, 4, 5]. Further investigations are needed to determine whether this new behaviour is caused by coherence side-effects or other new factors introduced by DAE execution on big.LITTLE or whether the shift to the new architecture of the Cortex-A15 and Cortex-A7 alone are responsible.
5.4 Lowering the access phase core frequency

All measurements above have been taken while running the A7 at its maximum core frequency of 1400 MHz. In addition to already providing a lower frequency than the A15 (which runs at 2 GHz), all cores on the SoC also support DVFS, allowing us to adjust the individual frequencies even further. With this, we have lowered the frequency on the LITTLE core to 600 MHz in intervals of 200 MHz and measured the performance impact for a selection of granularities of each benchmark. We have included the granularities with lowest overhead and two additional values that show a balance between overhead and IPC improvement. For libquantum we have chosen the lowest granularity, as it demonstrates how side-effects can change the performance impact of lowering the frequency in some cases.

While all cores on the Exynos 5422 support a minimum frequency of 200 MHz, a certain minimum speed is required even during the access phase in order to not slow down the execution. As figure 16 shows, choosing a lower frequency for the access phase does indeed affect performance. The fact that the observed performance impact
is not proportional to the frequency change gives the option to save additional energy at minimal performance cost for some frequencies.

As mentioned before, running on the big.LITTLE architecture we avoid switching the frequency between each slice, as we can clock the big and LITTLE cores individually. As soon as we select our ideal processor frequencies for access and execute phase, the cores can remain at these speeds for the entire execution.

The measurements also show that not all benchmarks and even individual granularities within the same benchmark are affected equally by the frequency change. This is due to the fact that the workload in access phases can vary based on how many address calculations and how much control flow the code needs to perform. But different frequencies and the resulting change in processing speed can also affect the previously described overlap of the two phases. This makes the LITTLE core frequency a good candidate be a parameter for a Software Multiversioned variant of future compiler passes.

5.5 Performance

Breaking down the time we spend inside the individual functions that contain the targeted loop, we can analyse which part is causing the increase in execution time. Figure 17 illustrates this for all three benchmarks.

Here we can clearly see that, while the execute phase gets faster, overhead is slowing down the overall execution as we reduce the granularity. This portion of the function time is dominated by the locking overhead between the two phases, as the
time to initialize and join the threads is insignificant (<1ms). As mentioned before, this overhead is proportional to the total number of slices, as each slice causes two locking operations. This makes large granularities perform significantly better. Unfortunately, the high amount of overhead also makes it difficult to make a meaningful comparison of our findings to previous DAE results.

While this overhead is affecting the overall runtime, we are also not speeding up the execute phase as much as previous DAE implementations. A major factor for this is the lack of a shared LLC on our system. Running the access phase on the A7 only brings in the data into the LITTLE cluster. As a result, touching prefetched data in the execute phase no longer results in an instant cache hit. The cache miss is just serviced by the A7 cluster instead of by main memory. Effectively this means that we are not making data instantly available in the cache, but are merely reducing the cache miss latency on the A15. The result is that we are now observing the full memory latency on the A7 and a reduced load time on the A15 instead of a much shorter combination of full memory latency in the access phase plus cache hit latency in the execute phase.

![Function runtime breakdowns](image)

(a) LBM breakdown  (b) CIGAR breakdown  (c) libquantum breakdown

Figure 17: Function runtime breakdowns

5.6 Energy savings

The previous individual results show that we managed to speed up the execute phase significantly for two out of the three benchmarks. While we were only able to do this at the cost of slowing down the overall execution time, it is important to consider that more than half of it is now spent on the LITTLE core as part of the access phase. During this time we not only run on a more energy-efficient microarchitecture
but also at a lower base clock frequency with the option to lower it even further at relatively small performance penalties.

With excessive overhead potentially nullifying all energy savings that we achieve through speeding up the execute phase, the current implementation of DAE for big.LITTLE requires us to find a balance between reducing the time we spend on the big core and keeping the overall runtime low. As we do not only observe notable IPC improvements for small granularities where locking overhead becomes problematic, but also for larger slices (e.g. 28% IPC speed-up at 1.77x slow-down compared to 37% peak improvement at 2.82x slow-down for CIGAR), the granularity effectively becomes main point of adjustment when we want to strike this balance.

The suboptimal performance that results from the lack of a shared LLC between the big and LITTLE core mentioned in the previous section also affects our energy consumption directly. Not being able to reduce the stalls in the execute phase further means that we are spending more time on the power-hungry big core than an ideal DAE implementation would.

5.7 Outlook: Optimizing the implementation

While only one of the suggested optimizations has reached the state where it could be applied as part of this work, we can already show that overlap and a timing-based DAE for big.LITTLE can indeed improve how well we perform.

The current implementation only allows us to adjust overlap in one direction. As our locking scheme always ensures that all prefetch hints have been issued before the execute phase starts, we can only delay the start of our execute phase and not start it early. However, as the non-blocking prefetches might still need time to actually load the data into the cache, adjusting the overlap still has the potential to be beneficial. In fact, preliminary results show that, for very small granularities, delaying the start of the execute phase in libquantum can yield an additional IPC improvement of over 6%.

Switching to a timing-based implementation, we can indeed noticeably reduce the overhead, and as a result the overall runtime. Initial tests have shown that we are able to bring slow-down in LBM down to 1.22 in comparison to our currently lowest value of 1.82. On the other hand, the current test implementation is not achieving the desired IPC improvements, as the two phases are most likely no longer aligned correctly and as a result do not access the data in the correct timings.

We have observed that, as predicted, common methods to suspend threads (such as calls to nanosleep) are not accurate enough for a purely timing-based approach and as a consequence can lead to suboptimal performance in a hybrid solution as well. This makes finding a reliable method to suspend the threads with the required accuracy an important prerequisite for a successful implementation.

6 Conclusions

The benchmark results show that Decoupled Access-Execute on ARM big.LITTLE can indeed provide noticeable energy savings, but only at the expense of sacrificing performance. As part of the development and evaluation process, we have identified the bottlenecks of the current implementation and suggest concrete optimization concepts for future iterations of this work.

Our prototypes successfully demonstrate decoupled execution on a heterogeneous architecture by running memory-bound sections on the energy-efficient LITTLE core
and compute-bound parts of the program on the performance-focused big core. For two out of three benchmarks we were able to improve the access phase IPC significantly (up to 37%), reducing the time they are executed at high frequencies and on performance-focused hardware.

Our experiments also make the main bottlenecks of the current implementation apparent. The locking overhead that has been introduced as a result of parallelizing the decoupled execution is the main reason for the overall benchmark slow-down. As it is proportional to the number of slices, choosing some granularities is no longer viable. Instead we are now facing a trade-off between IPC improvement and slow-down in the overall runtime. While the synchronization overhead plays a big role, we are also limited by the fact that the CPU clusters in the Exynos 5422 do not share a LLC. For our implementation this has the far-reaching disadvantage that any prefetches performed in the A7 cluster are not directly available for the A15 cores and have to be brought in through coherence instead. This prevents DAE to perform at its full potential.

While this problem is unavoidable on the current system, adding a shared LLC is a simple solution to it. In fact, newer interconnect designs already support a shared L3 cache that is directly connected to the bus. The AppliedMicro APM883208-X1 for example, while not a big.LITTLE chip, uses a shared L3 cache between clusters. Moving up another level in cache hierarchy and hosting big and LITTLE cores within the same cluster with a shared L2 cache would reduce latency even further.

It is also worth noting that the benchmarks evaluated as part of this thesis have not been designed or optimized for ARM architectures. As a result, potentially suboptimal memory access patterns and side-effects that not occur on previous analyses on x86 systems make it difficult to compare results between the two architectures side by side.

7 Future work

The first step in continuing this project is to translate the transformation pattern that was applied manually as part of this work into a set of LLVM compiler passes. This will allow us to apply Decoupled Access-Execute on ARM big.LITTLE to a wide range of benchmarks and general purpose applications and tie in this project with the current DAE work. In addition, the proposed optimizations need to be evaluated thoroughly and, once conclusive results are available, integrated to become part of DAE on big.LITTLE.

The previous work from Kamruzzaman, Swanson and Tullsen [9] explored using multiple cores for prefetching simultaneously, a concept that can be applied to DAE for big.LITTLE as well, as multiple LITTLE cores are often available on the same cluster. This could reduce the waiting time after each execute phase significantly in memory-heavy applications, as data can be prefetched at a much higher rate.

As Van Craeynest et al. [11] brought up, using purely memory- or compute-bound properties of a task to decide which core to execute it on is not always the ideal solution on heterogeneous architectures where the cores also perform differently based on available ILP, MLP and how the workload can benefit form out-of-order execution. The impact of this on ARM big.LITTLE needs to be investigated as well, as there is the potential to benefit from moving more parts of the original task to the LITTLE core.
8 References


