Efficient Execution Paradigms for Parallel Heterogeneous Architectures

KONSTANTINOS KOUKOS
Abstract

This thesis proposes novel, efficient execution-paradigms for parallel heterogeneous architectures. The end of Dennard scaling is threatening the effectiveness of DVFS in future nodes; therefore, new execution paradigms are required to exploit the non-linear relationship between performance and energy efficiency of memory-bound application-regions. To attack this problem, we propose the decoupled access-execute (DAE) paradigm. DAE transforms regions of interest (at program-level) in two coarse-grain phases: the access-phase and the execute-phase, which we can independently DVFS. The access-phase is intended to prefetch the data in the cache, and is therefore expected to be predominantly memory-bound, while the execute-phase runs immediately after the access-phase (that has warmed-up the cache) and is therefore expected to be compute-bound.

DAE, achieves good energy savings (on average 25% lower EDP) without performance degradation, as opposed to other DVFS techniques. Furthermore, DAE increases the memory level parallelism (MLP) of memory-bound regions, which results in performance improvements of memory-bound applications. To automatically transform application-regions to DAE, we propose compiler techniques to automatically generate and incorporate the access-phase(s) in the application. Our work targets affine, non-affine, and even complex, general-purpose codes. Furthermore, we explore the benefits of software multi-versioning to optimize DAE in dynamic environments, and handle codes with statically unknown access-phase overheads. In general, applications automatically-transformed to DAE by our compiler, maintain (or even exceed in some cases) the good performance and energy efficiency of manually-optimized DAE codes.

Finally, to ease the programming environment of heterogeneous systems (with integrated GPUs), we propose a novel system-architecture that provides unified virtual memory with low overhead. The underlying insight behind our work is that existing data-parallel programming models are a good fit for relaxed memory consistency models (e.g., the heterogeneous race-free model). This allows us to simplify the coherency protocol between the CPU – GPU, as well as the GPU memory management unit. On average, we achieve 45% speedup and 45% lower EDP over the corresponding SC implementation.

Keywords: Decoupled Execution, Performance, Energy, DVFS, Compiler Optimizations, Heterogeneous Coherence

Konstantinos Koukos, Department of Information Technology, Division of Computer Systems, Box 337, Uppsala University, SE-75105 Uppsala, Sweden.

© Konstantinos Koukos 2016

ISSN 1651-6214
ISBN 978-91-554-9654-8
urn:nbn:se:uu:diva-300831 (http://urn.kb.se/resolve?urn=urn:nbn:se:uu:diva-300831)
Dedicated to my very supportive wife,
for tolerating me those five —full of deadlines— years.
Also, dedicated to my son,
for all these sleepless —sometimes productive— nights.
List of papers

This thesis is based on the following papers, which are referred to in the text by their Roman numerals.

DOI=http://dx.doi.org/10.1145/2464996.2465012

DOI=http://dx.doi.org/10.1145/2544137.2544161

DOI=http://dx.doi.org/10.1145/2892208.2892209

DOI=http://dx.doi.org/10.1145/2889488

Reprints were made with permission from the publishers.
Other publications not accounted in the thesis:


Contents

1 Introduction .................................................................................................. 9
  1.1 The end of Dennard scaling ........................................................... 10
  1.2 Compiler challenges to automate DAE ........................................ 11
  1.3 Extend, adapt, and apply to general-purpose applications ............ 11
  1.4 Improve the programmability of heterogeneous systems ............ 12
  1.5 Thesis organization ........................................................................ 13

2 Understanding the decoupled access-execute paradigm ......................... 14
  2.1 The key idea ................................................................................... 14
  2.2 Task-based DAE; motivation and methodology ........................... 16
  2.3 DAE case-study .............................................................................. 17
  2.4 Evaluation ....................................................................................... 18

3 Compiler support for the decoupled access-execute paradigm .............. 20
  3.1 Affine code transformation ............................................................ 21
  3.2 Non-Affine code transformation ................................................... 22
  3.3 General purpose code transformation ........................................... 23
  3.4 Evaluation ....................................................................................... 23

4 Multi-version decoupled access-execute (SMVDAE) ............................ 25
  4.1 SMVDAE definitions and methodology ....................................... 26
  4.2 SMVDAE code generation ............................................................ 28
  4.3 Evaluation ....................................................................................... 29

5 Building heterogeneous UVMs without the overhead ............................ 33
  5.1 Synchronization and memory consistency model ........................ 34
  5.2 VIPS-G, heterogeneous system coherence ................................. 35
  5.3 Optimizations for GPGPU computing .......................................... 36
  5.4 Evaluation ....................................................................................... 38

6 Related-Work ............................................................................................. 40

7 Conclusions ................................................................................................ 42

8 Svensk sammanfattning ............................................................................. 44

9 Acknowledgments ..................................................................................... 46

References ........................................................................................................ 48
Computers nowadays, are inextricably linked to science and to our every-day life. Although the digital revolution of the last century has affected almost every other science, some scientific fields own their exponential development in those marvelous machines. High performance computing have become a necessity today for fields like biology and pharmaceutical research, physics and cosmology, economics, and forecast. Even in our every day life, we are surrounded by computers not only to browse the world-wide-web and our e-mail, but also for social media, entertainment and communication. Furthermore, numerous other devices (e.g., digital cameras, microwaves, cars, etc) contain at least one embedded processor. Even mobile phones and tablets today, are build with such sophisticated processors that easily outperform the capabilities of desktop computers of the previous decade. This widespread-use of computers comes with a surprisingly large energy footprint; it is estimated that computers today are responsible for 10% of the total world’s electricity consumption [52].

Since we rely so much on computers, significant effort from computer architects and engineers is dedicated to deliver higher performance and improve the energy efficiency of every new computer generation. This effort is two-fold: (i) improve the memory system, and (ii) exploit the opportunities to save energy based on the program demands.

The memory system [29] as evolved in the last years, refers to a memory hierarchy that consists of several levels of caches between the processor and the main memory (DRAM) —while it extends to the disk and non-volatile storage memories that are outside the focus of this study. Each level of the hierarchy has different characteristics such as: size, access-latency (smaller caches are faster), data-transfer energy, and meet different technology trends and costs (with registers and smaller caches being more expensive to produce per byte). Ideally, to exploit the full potential of state-of-the-art micro-architectures we would like to keep the data as close as possible to the execution units. Unfortunately, real life applications have a significantly bigger memory footprint than the processor caches, which makes it impossible to keep all data close to the execution units and therefore, they stress the performance gap between the processor and the main memory.

In many cases the memory system limitations, throttle the performance of applications with relatively big memory footprints, especially when they perform irregular memory accesses; for brevity we refer to them as memory-bound applications in the rest of this thesis. For these applications, dynamic
voltage-frequency scaling DVFS [24, 8] has been the staple of energy saving over the past years. DVFS provides quadratic energy savings for at most linear performance degradation\(^1\), and can be applied at different levels — for example, at system level with the help of the OS, at program level with assistance from the compiler, or entirely at the hardware level [34]. The software *decoupled access-execute* (DAE) technique proposed in this thesis (Paper I) [39], operates at program level with assistance from the compiler (Papers II & III) [31, 40]. This work is detailed in Chapters 2–4. The rest of the current chapter discusses the challenges that this thesis is addressing.

1.1 The end of Dennard scaling

In early 2000s the increasing frequency trend of the previous decades started to faint, as a result of materials properties that make frequency scaling beyond 4GHz at a reasonable power/thermal budget really difficult. This led to the introduction of multi-cores, because there was still room to scale the transistors (according to Dennard’s scaling). The underlying insight behind Dennard scaling [14] was that transistors can get smaller with constant power density, maintained due to a linear voltage and current decrease. Dennard scaling, however, comes to end due to the exponential increase of leakage power at low voltages, which restricts voltage scaling and undermines the effectiveness of DVFS. With voltage scaling range expected to shrink even more in future processor generations [28], optimizing the program execution for energy efficiency becomes a great challenge.

Frequency scaling (decrease) alone, provides linear power decrease at the cost of linear performance loss, which restricts the potential for energy savings in predominantly compute-bound applications. The performance of memory-bound applications, however, is not linearly affected by frequency scaling, because they spent most of their execution time waiting for data to come from the main memory. Therefore, their performance is limited by the memory, which allow us to decrease the CPU frequency with a small performance degradation while achieving near-linear energy savings. Unfortunately, real life applications are neither entirely compute- nor memory-bound. Instead, they consist of various compute- to memory-bound phases, where each has its own optimal energy efficiency at a different frequency. Additionally, the DVFS transition-latency for the hardware incurs overheads that prohibits its use at very fine-granularity (instruction-granularity) even in future processors with on chip voltage regulators [37, 38].

To attack these problems, we propose a software *decoupled access-execute* (DAE) paradigm (Paper I) [39]. They key idea of DAE is to transform the execution in two distinct, coarse-grain phases: one predominantly memory-bound

\(^1\)\(\text{Power} \propto ACfV^2\) where C is the load capacitance, V is the supply voltage, A is the activity factor and f is the operating frequency. \(AC\) is referred as effective capacitance \(C_{eff}\).
phase intended to prefetch the data to a private cache, and another compute-bound that will use the already prefetched data in its execution. With those phases designed to operate at a coarse granularity (task, or function), a low latency (nano-scale) DVFS can optimize the energy efficiency of each one separately with negligible overhead. Furthermore, our technique increases memory level parallelism (MLP), and therefore led to performance improvements (up to 15%) on irregular, memory-bound applications (as shown in Paper I).

1.2 Compiler challenges to automate DAE

The DAE technique discussed in Paper I [39], offers a great potential to improve the energy efficiency (on average 25% energy-delay product [17] improvements), while at the same time maintains the performance of applications, as if they were running at the highest frequency. There was an obstacle, however, for this technique to be generally applicable to all types of applications. That was the lack of a fully-automated way to transform them into DAE.

To overcome this limitation and allow automatic DAE transformations (initially of task-parallel applications) at task granularity we develop an LLVM [47, 48] compiler pass as proposed in Paper II [31]. The main goal of Paper II [31] is to relief the programmer from the responsibility to manually craft the access-phase for the DAE execution; instead the compiler was automatically generating an access-phase for each task, and properly integrate it into the program’s execution. This poses new challenges for the compiler; how to handle different codes, such as affine of non-affine codes? How to automatically generate an access phase that introduces minimal overhead due to prefetch instructions, and yet maintains most of the effectiveness of DAE for different classes of applications?

1.3 Extend, adapt, and apply to general-purpose applications

The decoupled access-execute model has a great potential to improve energy efficiency in the multi-core era, yet as we move forward to heterogeneous systems, the programming models evolve to take advantage of the heterogeneity of those systems, which integrate only a few sophisticated (and power-hungry) CPU cores, with many simpler (and energy efficient) GPU cores. The uprising use of general-purpose GPU (GPGPU) computing in many fields and applications creates new challenges. Since most of the regular code can be easily parallelized and run in a GPU or an accelerator, what is left for the sophisticated CPU cores is highly irregular regions of code [3]. The situation is aggravated by the fact that in heterogeneous systems the CPU cores might be
sharing resources (such as the memory bandwidth) with the GPU or the accelerators. This might further reduce their energy efficiency since the memory hierarchy becomes slower as it gets more congested.

Heterogeneity is not only a great challenge for the computer architecture, but also for the programming models and the compilers. To address this upcoming challenge, in Paper III [40] we extend DAE methodology (and the compiler infrastructure) for general-purpose applications. This introduces new challenges for the compiler that now has to generate efficient access phases for loops or basic-blocks with complex control flow, indirections and pointer chasing. Furthermore, we consider that the optimal energy efficiency may vary depending on dynamic information (such as execution-conditions and system utilization); therefore, a static-only approach can exploit only partial benefit. To address these issues we propose multi-version DAE (Paper III) [40]. Multi-version DAE uses a combination of runtime/compiler support, where the compiler generates different access phases—with different indirection depths each—and the runtime system instruments and selects the best performing access phase for each application, for the given execution\(^2\) (taking into consideration the system utilization—for example, sharing last level cache and memory bandwidth).

1.4 Improve the programmability of heterogeneous systems

The decoupled access-execute methodology discussed in this thesis, is intended to improve the energy efficiency (and in some cases the performance) of homogeneous many-core systems, as well as one end of heterogeneous systems—that is, the latency aware CPU cores. The energy inefficiency in these systems is a result of the performance gap between the memory system and the processing unit, which creates stalls during the program execution on irregular or memory bound applications. On the other end of heterogeneous systems we typically have a GPU with limited general-purpose computing capabilities. The GPU is highly optimized and specialized for graphics, while its general purpose computation capabilities are limited to a data-parallel execution model—for example, CUDA [53] or OpenCL [73]. GPUs are throughput oriented devices with higher bandwidth compared to CPUs, and their architecture and execution model are capable of hiding most of the stalls of the memory system. This is achieved with scheduling mechanisms (provided by the hardware) that allow very fast switching between idle and ready to execute threads. Since there is always work to be executed in the GPU’s execution

\(^2\)This study is focused on the compiler side; therefore, it is limited in automatically generating all possible different access phases, and omits an extended study of the runtime integration and overheads.
units due to these mechanisms, the potential of running DAE on the GPU is very limited. Therefore, for this part of the heterogeneous system, we shift our attention on how to improve the programmability rather than the execution efficiency, which is already highly utilized.

One of the main limitations of heterogeneous systems is the ease of use, with explicit data copies between host and device memories and synchronization, being a heavy burden for the programmer. Both industry (e.g., HSA foundation members [42, 26]) and academia [58, 60, 70, 21, 20] envision future heterogeneous systems to employ a unified virtual memory (UVM) across devices. The main design-challenge in a heterogeneous system with UVM is to optimize it to maintain the latency sensitive nature of CPU cores even when shared resources are under heavy stress due to high GPU utilization, especially when cache coherence is employed system-wide. Another challenge, is to provide an energy efficient address translation scheme for the GPU with respect to the data parallel execution model, and the massively multi-threaded architecture of this device. Therefore, having a private TLB per compute unit\(^3\) might not be the most energy efficient approach. To address these issues, in Paper IV [41] we propose a heterogeneous system architecture capable of providing UVM system-wide, with respect to the execution nature of the application on each device (latency sensitive for the CPU, throughput oriented for the GPU). In our approach we: (i) adopt the heterogeneous race free (HRF) [25] memory model while keeping synchronization semantics backwards compatible, (ii) propose an energy efficient GPU memory management unit (MMU), and (iii) provide hardware coherence for the entire system.

1.5 Thesis organization

The rest of this thesis is structured as follows. Chapter 2, explains the decoupled access-execute paradigm, discusses its applicability, its limitations, and evaluates a proof of concept on real systems using task-parallel applications with hand-tuned access phases. Chapter 3, details our compiler approach to transform different types of code to DAE and evaluates the compiler efficiency in this task. Chapter 4, explores the potential of multi-version DAE. With multi-version DAE capable to adapt in different system loads and handle general purpose code on the CPU, we shift our attention to heterogeneous architectures. For that Chapter 5, explores the design space to provide unified virtual memory in future heterogeneous systems, with minimal overheads for the coherence and the address translations for the GPU. Chapter 7 summarizes Chapters 2–5, while Chapter 6 discusses the related-work. Finally, Chapter 8 provides a summary of this thesis in Swedish (Svensk sammanfattning).

\(^3\)Compute unit is a GPU core, using OpenCL terminology. NVIDIA’s terminology use the term Stream Multiproccessor (SM).
2. Understanding the decoupled access-execute paradigm

One of the most widely used energy saving techniques of the last decades: DVFS, is losing most of its effectiveness due to the end of Dennard scaling, as already discussed in Chapter 1. To address this challenge, and considering that future processor generations will feature low latency, per-core DVFS [37, 38], we propose a novel execution paradigm: the *decoupled access-execute* (DAE) paradigm. Our first approach on DAE (Paper I [39]) targets task based applications (as explained in this chapter). This approach was later on extended to automatically handle affine and non-affine codes with the help of a compiler (Paper II [31]), and target general purpose applications (Paper III [40]) —as explained in Chapters 3 and 4.

2.1 The key idea

To provide a deeper understanding of our technique, Figure 2.1 demonstrates an execution example of a *coupled* (CAE) versus a *decoupled* (DAE) execution. The coupled execution is the original program execution that contains memory accesses mixed with arithmetic operations that use the data when they arrive from the memory hierarchy. Diagrams (a)–(c) in Figure 2.1 show the processing-unit utilization under different frequencies. When a data-access (*load instruction*) misses in the cache the processing-unit may stall, if it does not have enough outstanding, independent instructions to execute.

*Figure 2.1. Example of coupled execution under different core frequencies (a,b and c) and decoupled execution (d)*
In cases (a) and (b) a long latency load (LLL) causes the execution unit to stall. Although it does not produce any useful work while stalled, the execution unit keeps consuming power for the corresponding frequency. Reducing the frequency, however, reduces the power consumption, but it also makes the arithmetic instructions execute slower. Therefore, it can be predominantly beneficial only for program phases that have a lot of stalls, while it harms the performance (and in some cases even the energy efficiency) of computationally intensive programs or program-regions.

Figure 2.1.c shows an example of inefficiently scaling-down the frequency. In this case some stall remains, while for the computationally intensive regions the performance is degraded. An ideal DVFS implementation, would try to set a low frequency for the memory-bound regions and instantly switch to a high frequency for the compute-bound regions of the application. The DVFS transition latency, however, prohibits this technique at very fine granularity (a few instructions); instead state-of-the-art DVFS techniques select an intermediate frequency, which offers a compromise between performance losses and energy savings. To overcome this limitation and maintain both good performance (as of the highest frequency) and good energy savings (as if we could apply ideal DVFS) we propose DAE, which transforms the code in two coarse grain phases that we can independently set optimal frequencies\(^1\).

The first one; the access-phase is a skeleton of the original code-region that maintains only the load instructions and converts them into software prefetch instructions (at the deepest indirection level). All stores and arithmetic instructions (with the exception of address calculations) are removed to create a side-effect free phase. This phase is intended to prefetch all the data that the next phase (the execute-phase) will use. Therefore, all of the stalls that a normal execution would have, are eliminated within the access-phase, which makes it a good candidate for execution at a low frequency.

The second one; the execute-phase is the original (unmodified) region that runs “hopefully” without stalls, since the vast majority of long-latency misses are resolved in the previous phase. Therefore, this phase runs at a high frequency and maintains an overall high performance to the application, while the energy savings comes from the access-phase that runs on a lower frequency (with a minimal impact on performance). Overall, DAE achieves both high performance (as if the application was running entirely at a high frequency) and good energy savings (as if the application was running at an optimal frequency).

\(^1\)Our approach requires low transition-latency for the frequency and per-core DVFS. Research proposals envision future systems to feature such characteristics. In Paper I [39] we find that when DAE is applied in a system with a DVFS transition latency below 500ns, provides good energy savings for memory bound applications.
2.2 Task-based DAE; motivation and methodology

The wide-spread use of multi-core and many-core systems of the last decade, necessitates a new approach to efficiently program those inherently parallel machines. Parallel programming, has practically been proved a challenging task for the programmer, especially for high performance computing. This challenge led computer scientists to develop task-based programming models that reduce the complexity of typical parallel programming environments (e.g., POSIX threads and message passing MPI). Several frameworks have been developed over the last years such as: StarSs [56, 4], SMPSS [51], Intel TBB [55], Cilk5 [16], TPC [76], etc. The main advantages of task-based programming environments are:

- Their ability to simplify synchronization.
- Their load balancing (e.g., work-stealing) capabilities.
- They provide abstractions to simplify heterogeneous programming.

Therefore, in Papers I [39] and II [31], we adopt the task based programming model for DAE. In addition to the advantages described above, a task-based model allow us to adjust the data granularity of each task, and also provides a well defined scope (task-scope) to apply DAE. Therefore, by carefully adjusting task granularity to make its dataset fit in a private cache we are able to set the DVFS region limits to a coarser granularity, which incurs negligible DVFS transition overhead.

In our approach we are using a custom runtime system that implements each task as a C/C++ function. The role of the runtime is to orchestrate the parallel execution; that is, to allow for task synchronization, and take care of the task scheduling and load balancing. Our runtime is kept minimal and omits support for sophisticated point-to-point synchronization between tasks (we also omit dependency analysis and rely on simple barriers, which leads to a data race free execution for the tasks within each epoch). This decision helps us minimize the runtime overhead and allow the execution of fine-grained tasks.

Furthermore, the runtime system is responsible for profiling the tasks and adjusting the frequency of the access- and execute-phases of each task. To determine the optimal frequency for each phase, we employ a linear to IPC power model in combination with online time profiling of each phase of the task (as described in Paper I [39] Section 3.3). The role of the power model is two-fold: (i) provides an energy estimate per phase, even when different tasks are scheduled in different cores and their execution overlaps, and (ii) allow us to predict the optimal frequency for each phase of the task (given a hardware specific power model), and therefore to optimize either for energy, or for energy-delay product (EDP). To validate the accuracy of our model we compare the total power estimate against the measured value using the methodology described in [39], and [72], with an average error of $< 5\%$.
2.3 DAE case-study

Figure 2.2. The impact of core frequency on execution time and energy for CG with 1, 2 and 4 threads. For CAE we use frequencies from $f_{\text{min}}$ (1.6 GHz) to $f_{\text{max}}$ (3.4 GHz) with a step of 400 MHz while for DAE we use $f_{\text{min}}$ for the access-phase and $f_{\text{min}}$ to $f_{\text{max}}$ with a step of 400 MHz for the execute phase.

Figure 2.2 shows the execution time and energy of a decoupled (DAE) as opposed to a contemporary (CAE) execution for different frequencies and number of threads. In this case study we use the minimum frequency (1.6 GHz) for the access-phase of DAE execution and vary the frequency of the execute-phase, similar to the corresponding CAE execution. We observe that DAE maintains the performance of the corresponding CAE execution on the same frequency despite running the access-phase at the lowest frequency. An one-to-one comparison between DAE and CAE shows no performance degradation in any combination of frequency and number of threads by DAE. The key observation of Figure 2.2 is the energy efficiency as the frequency scales. Since a significant amount of the execution-time is spent within the access-phase, as we scale-up the frequency for the execute-phase (and therefore the performance) the potential for energy savings in DAE increases. In case of 4 threads at the highest frequency we observe around 25% energy savings over the corresponding CAE execution (with exactly the same performance).
2.4 Evaluation

![Bar chart showing normalized execution time and EDP of CAE versus DAE for Intel Sandybridge using 4 threads (baseline is CAE at maximum frequency).]

Figure 2.3. Normalized execution Time and EDP of CAE versus DAE for Intel Sandybridge using 4 threads (baseline is CAE at maximum frequency).

The decoupled access-execute paradigm provides a unique potential to improve energy efficiency without compromises in performance. Figure 2.3 shows the execution time and EDP for different applications\(^2\). For CAE we evaluate the performance and EDP for two different frequencies: maximum and optimal-EDP (where maximum-frequency CAE is used as baseline), while for DAE we evaluate a min/max and an optimal-EDP policy. The min/max is a naive policy that runs the access-phase in the minimum and the execute-phase at the maximum frequency. However, this might not be the optimal neither for the access-phase because sometimes it might contain complex address calculations that benefit from a slightly higher (than the min) frequency, nor for the execute-phase because there might be some remaining stalls, and therefore the optimal-EDP is at a slightly lower frequency (than the maximum). For that, we

\(^2\) EDP [17] is the product of energy by time. It emphasizes energy saving with respect to performance \((EDP \propto C_{eff} f V^2 T^2)\) where: \(C_{eff}\) is the effective capacitance, \(f\) is the frequency, \(V\) is the voltage and \(T\) is the execution time.
integrate the power model in the runtime-system to predict the optimal-EDP frequency (based on each phase’s IPC).

Figure 2.4. Simulation of future DVFS transition latency using 4 threads for the average of memory bound versus compute bound applications

Performance-wise not only we do not introduce any degradation, but on the contrary; for memory bound applications we achieve improvements of up to 15% due to the MLP\(^3\) improvement of the prefetching nature of DAE. In terms of EDP the benefit is two-fold; DAE improves the EDP both due to the overall performance improvement, and due to the energy savings of the access-phase. For the memory bound applications we measure EDP gains of up to 50%, while on average we have an \(\approx 25\%\) improvement. This study however, assumes a very-fast (almost instant) DVFS transition latency\(^4\).

Figure 2.4 models the impact of DVFS transition latency on DAE for different classes of applications. Memory-bound applications have a great potential for energy savings, but they are also more prone to DVFS transition latency because the duration of each of their phases is rather fine-grain. In contrast, compute-bound applications have a potential for slight benefit, but they are also less affected by the transition overhead because the duration of their execute-phase is rather coarse-grain (such applications typically involve floating point arithmetic). For the average of all applications we observe that a DVFS transition-latency below 500ns allows for energy and EDP improvements to memory-bound applications, without harming the energy or EDP of compute-bound applications. For contemporary systems that do not feature low-overhead, per-core DVFS, DAE can be used with disabled DVFS functionality. This maintains a fraction of the benefit for memory-bound applications; that is, performance improvements (and the EDP saving that derives from them) due to increased MLP.

\(^3\)Memory-level parallelism (MLP) is the architecture ability to support multiple outstanding memory operations.

\(^4\)The evaluation methodology is detailed in Section 3.4 of Paper I [39]
3. Compiler support for the decoupled access-execute paradigm

The decoupled access-execute paradigm relies its efficiency on the access-phase to prefetch the right data while keeping the instruction overhead minimal. A straight-forward approach to generate the access-phase is to create a lightweight skeleton of the execute-phase that omits all unnecessary arithmetic calculations and stores. To maintain correctness the access-phase must be side-effect-free. Therefore, stores to variables outside the task-scope, or function-calls that directly or indirectly modify values outside the task-scope are forbidden in the access-phase. In Paper I [39], we used manually crafted and optimized access-phases based on these guidelines.

The challenge of Paper II [31] is to fully automate this procedure with support from the compiler (e.g., as an LLVM pass [47, 43]). Conceptually, the use of a prefetching access-phase before the actual task increases the number of instructions and adds overhead to the total execution. This overhead is compensated by the speedup of the execute-phase. However, there is no guarantee that the total execution time (both access and execute phases) in DAE will be smaller or similar to the original (CAE) execution time. To ensure that DAE will not penalize the total execution time due to excessive instruction overhead in the access-phase, the latter must be kept instruction-minimal with respect to the following:

- **No duplicate address prefetching (or calculation).** The data used in the execute-phase must be prefetched exactly once in the access-phase even in deep loop nests with high temporal locality. Similarly, address calculations should also be kept minimal, which suggests loops transformations to reduce the original loop nest in some cases (e.g., affine-codes)
- **No store prefetching.** Empirically we found that stores are less critical to create stalls in the pipeline; therefore, we decided not to prefetch them to keep the access-phase more lightweight.

Although Paper I [39] targets task parallel applications, DAE is applicable to various types of codes. Our effort to build a compiler infrastructure for DAE starts with Paper II [31], in which we target affine and non-affine codes similar to Paper I and extends to Paper III [40] in which we add support for general-purpose codes (e.g., SPEC-CPU (2006) [22]). The current chapter (Sections 3.1–3.2) details the transformations presented in Paper II [31], and introduces (Section 3.3) the approach of Paper III [40], which is detailed in Chapter 4.
3.1 Affine code transformation

Affine code transformations apply to loop nests with static control structures, where data accesses can be represented as affine functions of the enclosing loop indices. Typical examples of such codes are linear algebra codes (e.g., matrix multiplication, LU, Cholesky, etc). For these codes, we employ the polyhedral model [6, 67] (available in LLVM through PolyLib [57]) to generate the access-phase. The polyhedral model is a powerful mathematical model that provides a geometric representation of software loops. This allows the compiler to reason about complex loop transformations such as loop unrolling, fusion, fission, and parallelization. In our compiler pass for DAE, the polyhedral model is used to determine the set of unique memory addresses accessed by each task. Therefore, it allows us to generate an access-phase with the minimal loop nest to prefetch all the addresses accessed in the task.

Listing 1 Loop nest accessing different arrays

(a) /* Execute version */
   for (i = 0; i < Block; i++)
      for (j = i+1; j < Block; j++)
         for (k=0; k < Block; k++)
            A[j][k] = D[j][i] * A[i][k];

(b) /* Access version */
   for (i = 0; i < Block; i++)
      for (j = 0; j < Block; j++)
         prefetch: A[i][j]
         prefetch: D[i][j]

Listings 1 and 2 and show an example of loop nests accessing different arrays or different tiles on the same array and their corresponding access-phases. In Listing 1 the original task is accessing tiles on different arrays (A and D). Therefore, we first compute the convex hull for each array and then merge them into the same loop nest (only if they have the same number of iterations) as shown in the second code-region of Listing 1. Otherwise, we generate two separate prefetch loop nests.

Listing 2 illustrates another case where the original task is accessing different tiles on the same array. In this case a convex hull would include (in

1Paper II [31] discusses the details of each case illustrated in Listings 2 and 1. This thesis gives a high-level overview and omits technical details.

2One may observe that for array D only the elements below the diagonal are accessed, while the loop bounds are the same as with array A. This creates a dilemma whether we should generate prefetches for the upper part of array as used in Listing 1 or use separate prefetch loop nests for A and D. Through experimentation we find that a single loop nest has less overhead, and therefore use it in the rest of Paper II and this thesis.
Listing 2 Loop nest accessing blocks of the same array

(a) /* Execute version */
for (i = 0; i < Block; i++)
    for (j = i + 1; j < Block; j++)
        for (k = i + 1; k < Block; k++)
            A[Ax+j][Ay+k] = A[Dx+j][Dy+i] * A[Ax+i][Ay+k];

(b) /* Access version */
for (i=0; i<Block; i++)
    for (j=0; j<Block; j++)
        prefetch: A[Ax+i][Ay+j]
        prefetch: A[Dx+i][Dy+j]

addition to the blocks) the entire area between the accessed-blocks. To overcome this excessive prefetching we separate the memory accesses into classes with the same parameters (e.g., Ax and Ay for class A, and Dx and Dy for class D). Similar with the previous case we create two separate convex hulls and merge them only if the loop bounds are the same, as shown in the last code-region of Listing 2. This approach minimizes the instruction overhead, while it allows for prefetching of the exact address ranges.

3.2 Non-Affine code transformation

For non-affine codes, which are non amenable to polyhedral optimizations we use simple heuristics to generate the access-phase. We start this approach by generating a simplified skeleton of the original code that is further optimized to produce acceptable results in a wide variety of access patterns following the guidelines discussed in the beginning of this chapter.

One of the major roadblocks to generating efficient access-phases is that maintaining the entire control flow graph (CFG) may result in an access-phase very heavyweight, which replicates a lot of the original computation. This is very common when there is a control flow data-dependency. In this case we either have to preserve the entire CFG part required to calculate the control flow statements, or exclude the entire block from the access-phase. Unfortunately, there is no rule of thumb for this dilemma. In Paper II [31], we find through experimentation that keeping the access-phase minimal yielded better results for the set of evaluated applications.
3.3 General purpose code transformation

Although a naive, lean access-phase approach for non-affine codes as the one used in Paper II [31] yields sufficient results for scientific applications, it is not suitable for all types of general purpose code (e.g., SPEC-CPU (2006) [22]). In fact, there is no single remedy for the wide variety of access patterns and control flow graphs met on general purpose applications. Therefore, the compiler must consider the trade-off between address-calculation overhead and prefetching effectiveness.

Furthermore, various architecture-specific parameters (such as cache size and memory bandwidth) and non (such as system utilization) affect the benefits of prefetching. To address these issues, Paper III [40] proposes a software multi-versioning scheme, which uses indirection-measure as the prefetch throttle. By controlling the number of indirections for the generated prefetch instructions we are able to selectively exclude heavy-weight prefetches (with deep indirections), whose overhead outweigh the benefit. However, transformation for these codes is closely intertwined with multi-versioning; therefore, they are detailed in Chapter 4.

3.4 Evaluation

Our primary concern when automatically generating access-phases for the DAE paradigm is the instruction overhead. A small instruction-overhead can be easily tolerated in contemporary CPUs, especially when stalls exist—as in the cases that DAE targets. Excessive overhead however, may result in a severe performance degradation and reduce the potential for energy savings. Therefore, our first concern is to ideally maintain the performance of a CAE execution at highest frequency, or at least introduce a minimal degradation as a trade-off for the energy/EDP savings. Figure 3.1.a shows the normalized execution time for different execution modes such as: CAE at its optimal EDP frequency, hand-crafted (manual) DAE both with min./max. and OptEDP policies, and compiler generated DAE (both min./max. and OptEDP).

Overall, DAE applications with auto-generated by the compiler access-phases perform similar or even better than manually-crafted ones. On average a naive DVFS policy (min./max.) introduces a minimal performance degradation $\approx 1 - 2\%$ (worst-case up to 10%). Figure 3.1.b shows the energy savings from DVFS between DAE (hand-crafted versus compiler-generated) and CAE. Our first observation is that compiler-generated DAE slightly outperforms the hand-crafted versions, while CAE achieves around 7% better energy savings at the cost of 15% performance degradation. In terms of EDP as shown in Figure 3.1.c, we achieve the same EDP with CAE at its optimal frequency ($\approx 25\%$ lower EDP) but with significantly higher performance (on average 15% faster).
Figure 3.1. Performance, Energy, and EDP results on a quadcore Intel Sandybridge, assuming state-of-the-art 500ns frequency scaling transition latency.
4. Multi-version decoupled access-execute (SMVDAE)

General purpose codes (e.g., SPEC-CPU (2006) [22]) is the most challenging class of applications to optimize for DAE. Their control flow graph is more complex than typical scientific or linear algebra codes, with data dependencies in conditional statements, while their memory access-pattern may involve pointer chasing or deep indirections. Such access patterns are amenable to prefetching, since they create a lot of stalls; however, we cannot statically determine how much address calculation overhead can be tolerated on each application.

To attack this problem we propose a software multi-version decoupled access-execute (SMVDAE) scheme (Paper III [40]). In this approach we statically generate different access-phases, with different number of indirections —for brevity we call them access-versions — and we select the best-performing version at runtime. This technique not only allow us to optimize codes with statically unknown prefetching overhead, but also allow us to optimize the application under different system utilization. For example, in a multicore system where a single-threaded application runs in isolation (not sharing memory bandwidth or last level cache) a more lightweight prefetching DAE scheme may yield the best EDP, while if the same application runs in parallel with other processes (different applications, or instances of the same application), a deeper level of indirection in the access-phase could yield better results.

This adaptation is particularly useful in heterogeneous systems, where the CPU may share the same memory with the GPU, which generates tremendous pressure to the memory system. Furthermore, with the increasing offloading of easy to parallelize code to GPUs and accelerators in the heterogeneous era [3], CPU is expected to run only the complex and memory-bound parts of the applications. This necessitates an adaptive, dynamic approach as the one we propose in Paper III [40]. Our approach is not only adaptive (selects the best access-version when DAE is able to provide benefits), but is also self-healing —that is, it reverts back to CAE when it detects that DAE does not provide the requested benefits, so it always runs the application in its optimal mode.
4.1 SMVDAE definitions and methodology

\[
x = a [* b + c - d [* e)]; \quad t_1 = \text{load } b \\
t_2 = \text{gep } c, 0, \text{index of } d \\
t_3 = \text{load } t_2 \\
t_4 = \text{load } e \\
t_5 = \text{gep } t_3, 0, t_4 \\
t_6 = \text{load } t_5 \\
t_7 = \text{add } t_1, t_6 \\
t_8 = \text{gep } a, 0, t_7 \\
x = \text{load } t_8
\]

A load statement in C and its SSA representation, resembling the LLVM IR; “gep” is the GetElementPointerInst indexing instruction from LLVM.

Figure 4.1. The number of indirections of an address A is given by the number of loads required to compute A

SMVDAE provides us with a knob: the number of indirections, to control how light/heavy-weight an access-phase can be to provide benefits. We define number of indirections as a metric for the number of intermediate loads required to calculate the effective address for a memory location. Figure 4.1 shows an example of a load operation and its intermediate representation in LLVM (LLVM-IR). In this example a total of four intermediate loads \((t_1, t_3, t_4, \text{ and } t_6)\) are required to calculate the effective address of \(x\). Algorithm 3 summarizes the procedure to measure indirection-count.

**Algorithm 3** Finding the number of indirections for an address A

1. If address A is not produced by an instruction (e.g. a global variable), stop; the level of indirection is 0. Otherwise, continue.
2. Find the set \(D\) of dependencies for \(A\) (by following steps 2-4 in Algorithm 4, using \(D\) in place of \(K\)).
3. The level of indirection is the number of load instructions present in \(D\).

Figure 4.2 shows the overall system design. At user-level (source code) no modification of the application is required. The compiler is responsible to generate all different access versions up to a user defined maximum number of indirections. The programmer only needs to provide hints to the compiler for the region of interest (ROI) to optimize using DAE. This is a list of function names, which can be acquired through profiling. This list of functions is provided as a compilation parameter through an external text file. Furthermore, the compiler can perform more complex transformations for DAE such as loop chunking and apply DAE to each chunk separately. This ensures that each chunk’s data-footprint is small enough to fit in a private cache. Optionally, the programmer is allowed to select a fixed number of indirections for the entire execution or set the maximum indirection number.

After the compiler has generated all the available access-phase versions a runtime system is responsible to profile and orchestrate the execution. The
runtime consists of two parts; one part is system independent and is responsible for time profiling and version selection, while the other is platform specific and profiles the IPC through hardware counters, embeds a power-model to estimate execution energy, and optionally DVFS (when applicable) each DAE phase to its optimal frequency.1

---

**Figure 4.2.** The overall system design demonstrating all system components to support multi-versioning

---

For the DVFS runtime, we extend the linear to IPC power model used in our previous work of Papers I [39] and II [31] to support newest architectures

---

1In the implementation of Paper III [40], we focus our study on the compiler techniques to generate multi-version DAE. Therefore, we omit the details and the evaluation of a selection-runtime for future work. Instead, we perform a brute-force evaluation by generating all available versions and evaluate each one for the entire program execution.
(e.g., Intel Haswell [18]). Since the DVFS transition latency is very costly, we use a methodology similar to Paper I to emulate low-latency per core DVFS, while performance measurements are conducted in real hardware at the optimal frequency suggested by the power model. For the version selection we use a simple heuristic —that is, to select the best-performing access version with respect to the total access-execute runtime. For this we evaluate all access-execute pairs as shown in Figure 4.3 including a CAE run (only execute), once at the beginning of each loop and use that for the rest of the program execution. This approach trades off adaptivity to minimize selection overhead.2

4.2 SMVDAE code generation

Algorithm 4 Set $K$ collects the instructions preserving the CFG

1. Add all instructions that directly define the CFG to set $K$.
2. For each instruction $I$ in $K$, add all instructions that produce a value required by $I$.
3. If $I$ is a load instruction, add all store instructions that may store the value that $I$ reads.
4. Repeat steps 2 and 3 until no choice of $I$ will cause further instructions to be added to $K$.

Since multi-versioning allow us to select how light/heavy-weight the access-phase can be, the code generation for SMVDAE is entirely different from our prior work for non-affine codes of Paper II [31]. In contrast to our previous work, we do not employ heuristics to omit parts of the CFG in the access-phase with respect to their overhead (as in Paper II). Instead, we build a CFG skeleton with a minimal set of instructions that preserve the entire control flow graph of the original function, and adjust the overhead by controlling the number of indirections.

Algorithm 4 summarizes the methodology to collect the instructions to generate a CFG skeleton. The algorithm first adds all instructions that directly define the CFG (conditional/unconditional branches and jumps) into a set. Then it processes each instruction within the set by adding its dependent instructions (steps 2 – 3). Finally, the algorithm terminates when no new instructions can be added.

A major concern with this approach is to preserve a side-effect-free access-phase. For that, we perform an alias analysis using the built-in LLVM Alias-Analysis pass. If we detect that a store instruction aliases with a global variable or an address outside the scope of the access-phase, then the access-phase is considered non-safe and is discarded. Furthermore, the access-phase is con-

2 An extensive study of the selection-runtime overhead and adaptivity is left for future work.
sidered non-safe, when there is a CFG dependence to a non-read-only function call. The methodology to detect store aliasing is detailed in Paper III [40] (Section 3.1.1). Section 3.3 of the latter proposes different solutions to handle statically unknown memory dependencies.

4.3 Evaluation

![Normalized execution time and energy for all generated versions for mcf, and mri-g. The x axis displays the maximum number of indirections per version. Experiments were conducted on an Intel i7-4790K CPU.](image_url)

We begin the evaluation of multi-version DAE with a case-study of two applications: *mcf* and *mri-g*, as shown in Figure 4.4. In this study we show the benefit of multi-versioning for codes in which we cannot statically predict the best version (indirection count) for the access-phase. For each of the applications we observe that the number of indirections has a significant impact both on performance and energy efficiency. For *mcf* the best performance and energy efficiency is achieved with the maximum number of indirections in the access-phase (up to 3 indirections) —that is, 30% in performance and 45% energy. In the case of *mcf* all different access versions have an impact on performance and energy efficiency, while only one yielded the optimal results, exceeding by far all other available versions.
A different situation is observed for mri-g, in which the first three versions yield no benefit (neither in performance nor energy), while as we increase the number of indirections we improve both performance and energy. The optimal access version for this application uses 9 indirections\(^3\), and achieves $\approx 17\%$ performance improvement and $\approx 20\%$ energy savings.

\(\text{Figure 4.5.} \) Normalized execution time and energy for a selection of SPEC-CPU (2006) benchmarks, under different system loads on an Intel i7-2600K. 1P is 1 instance of the application in isolation, versus 4 instances (4P) of the same application (one per core).

Beyond optimizing the access-phase for codes that their optimal access-version cannot be statically known, SMVDAE, also allow us to optimize the

\(^3\)The optimal access version is not always the one with the more indirections. In mri-g it is the second before maximum indirections with small difference from the max-indirections version. In other applications it might be an intermediate version, although for brevity we exclude those in our case study of Figure 4.4.
access-phase with respect to the system load. Figure 4.5 shows the benefit provided by SMVDAE when an application runs in isolation, versus running multiple instances of the same application (that share resources). In Figure 4.5 we use a selection of memory bound applications from SPEC-CPU (2006) benchmark suite, where DAE is predominantly beneficial. The first observation in this figure is that DAE has a great potential for improvements (both in energy and performance) when the system is under heavy load (e.g., running 4 instances). Optimizing for different system loads separates the applications in two categories: (i) highly apt to DAE with respect to both performance and energy, and (ii) apt to DAE under heavy load with respect to energy savings.

In the first category we classify *mcf* and *milc*. These applications provide good performance improvements and energy savings when run in SMVDAE in both cases (1 instance and 4 instances). In the second category we classify *libQ, soplex, sphinx* and *lbm*. These applications do not provide performance improvements (with exception of *lbm* that yields a small speedup in 4 instances). However, they have a potential for energy savings when running under heavy system load (4 instances) because a significant amount of their execution time is spent within the access-phase, for which we scale down the frequency. On average for this selection of memory bound SPEC applications we have a $\approx 10\%$ performance improvement and up to $\approx 35\%$ energy savings.

We conclude this section by evaluating the majority of SPEC CPU^TM2006 in two different Intel architectures (i7-2600K and I7-4790K), as shown in Figure 4.6. The newer Intel processor (i7-4790K) has a slightly increased frequency, which augments the gap between the processing-unit and the memory system. Therefore, this architecture yields even more promising results for DAE. More specifically, we achieve over 20\% energy savings and $\approx 25\%$ lower EDP, while in the best case we achieve over 60\% lower EDP and energy savings. For the heavily compute-bound applications multi-versioning either reverts the execution to the original CAE to avoid hurting the performance due to excessive prefetching overhead, or select a very lightweight access-version, which yields minimal benefits.

The SMVDAE study (Paper III [40]) concludes our effort to optimize energy efficiency of many-core and multi-core CPUs. To summarize; so far in this thesis, we proposed decoupled access-execute paradigm and evaluated its performance and efficiency in real systems (e.g., Intel CPUs). We proposed a compiler methodology to automatically handle various classes of applications (affine, non-affine and complex general purpose). Finally, we propose a software multi-version DAE (SMVDAE), which allow us: (i) to optimize codes with statically unknown prefetching overheads and deep indirections, and (ii) to adapt under different system loads. The DAE paradigm yields very good energy and EDP savings due to its ability to optimize energy without compromises in performance. The next chapter proposes a novel, complete heterogeneous system architecture with unified virtual memory, to ease programmability and improve the overall performance and energy efficiency.
Figure 4.6. Overall performance, energy, and EDP improvements on Intel i7-2600K and i7-4790K
5. Building heterogeneous UVMs without the overhead

As already prefaced, the decoupled access-execute paradigm is not a good fit for GPUs, because their hardware is already designed to tolerate long latency stalls. Therefore, we do not provide a DAE version for GPUs; instead we try to improve the programmability of fused (CPU-GPU) heterogeneous systems. The main obstacle on these systems is the difficulty to maintain data coherent across devices. In a heterogeneous system without unified virtual memory (UVM), the programmer has to copy data back and forth to the device to maintain consistency. Furthermore, for this to be efficiently implemented the programmer should consider the benefits of overlapping data transfers with computation in the GPU, which makes this approach even more complicated.

Various research proposals [59, 58, 20, 70] recognize the need for UVM. However, there are two different tendencies based on the memory consistency model. The first, employs sequential consistency to provide UVM (e.g., Power et al. [59] and [58]), while the second adopts a more relaxed memory consistency, as in Hechtman et al. [20] and Singh et al. [70]. Our experimentation with the state-of-the-art simulator for heterogeneous systems [59], shows that using sequential consistency across a heterogeneous system introduces long access-latencies to the CPU when accessing data owned by the GPU. Furthermore, a naive directory-approach such as the one used in [59] suffers great performance losses due to directory and GPU’s last-level-cache contention (as quantitatively evaluated in Paper IV [41]). Therefore, we adopt a release consistency memory model, and more specifically the heterogeneous-race-free HRF [25] model. Our effort is many-fold and is outlined below:

1. **Ease of use.** A pointer is after all a pointer; therefore, our proposal employs unified virtual memory (UVM) between devices.

2. **Optimize each device with respect to its needs, and in harmony with the entire system.** The CPU is after all latency-sensitive; therefore, in our approach we try to protect the CPU from slow interactions with the GPU.

3. **Keep the architecture simple.** Employing sequential-consistency across an entire heterogeneous system, introduces a huge bottleneck: the directory and the last level GPU cache. Therefore, in our approach we use a directory-less coherence protocol, which simplifies the architecture.

4. **Optimize the GPU for general-purpose GPU (GPGPU) computing.** Finally, we optimize the write-policy of the GPU caches and the GPU memory management unit (MMU) for GPGPU applications.
5.1 Synchronization and memory consistency model

Figure 5.1. Different synchronization primitives of a heterogeneous x86/CUDA system.

GPUs today, use a weakly-ordered memory model to make writes visible in different levels of the memory hierarchy. This requires some effort from the programmer to define the synchronization scopes, but provides a potential to reduce NoC traffic since writes do not need to become visible to the entire system immediately. Therefore, explicit synchronization is standard in heterogeneous GPGPU computing today. Furthermore, state-of-the-art proposals such as the heterogeneous-race-free HRF [25] model, recognize the benefits of explicit synchronization.

In our system proposal we take into consideration both backwards compatibility (for legacy GPGPU codes) and future-system proposals (e.g., HRF). Therefore, in Paper IV [41] we adopt a relaxed-consistency memory model similar to HRF. Figure 5.1 shows a high-level overview of the synchronization scheme employed in our proposal. For the CPU, we use \texttt{pthread_barrier_wait()} or \texttt{pthread_mutex_lock()} /unlock() for synchronization within CPU cluster. These synchronization-primitives selectively-flush the first level cache of the cores involved (write-back the dirty-shared blocks, and invalidate all the shared blocks after writing-back). As long as these data are not required by the GPU, no further action is required in the system. To make these changes visible in the GPU, however, we facilitate the programmer with the \texttt{cpu_flush_llc()} primitive, which flushes of the last level CPU cache, making all updates within the CPU cluster visible in the entire system.

Similarly, GPU synchronization is explicitly declared by the programmer for each scope. For the GPU synchronization, we adopt the CUDA [53, 54] synchronization semantics (unmodified) in our system, as shown on the right-side of Figure 5.1. For example, \texttt{_threadfence_block()} is used to make writes performed by the calling thread visible to all threads in the thread-block. Sim-
ilarly, threadfence() ensures that writes become visible by all threads in the device, while _threadfence_system() makes them visible system-wide. Although these synchronization semantics are seemingly similar to the programmer, they incur different overheads, and their implementation faces different challenges (based on the system-components that participate), as explained in the next section.

5.2 VIPS-G, heterogeneous system coherence

Our starting point to efficiently support the HRF memory model, is by developing a coherence protocol that can truly address the needs of a heterogeneous system, and preserve the characteristics of each device (e.g., CPU is latency sensitive, while GPU is throughput-oriented). For that we develop a heterogeneous variation of the VIPS family of protocols [64, 35, 63]. The key idea of the VIPS protocol (and most of its variations) is to provide a simple, scalable protocol, that rely on explicit synchronization primitives to create race-free execution regions of the applications (epochs). The basic version of VIPS: VIPS-M [64], which we employ for the CPU coherence, is flat, directory-less and uses page-classification stored in the TLB and the page-table. Each page can be either private, shared, or invalid, which are the stable states of the protocol. For the private pages a write-back policy is selected to reduce unnecessary traffic to the lower-level, while for the shared pages a write-through policy is employed. To reduce traffic for shared blocks we use coalesced (delayed) writes.

Based on the VIPS protocol specifications blocks belonging to private pages can cross synchronization points without flushing. In contrast to them, shared blocks have to be written-back the next level of the memory hierarchy and self-invalidate. This guarantees that the data will become visible to other cores after synchronization and that the calling thread will not read stale data, since all shared data are invalidated and the new values are read from the next level of the hierarchy. In VIPS-M version [64] which we employ in our approach this classification is held at the TLB and the page-table. Such an approach is flat—that is, when a page becomes shared it is considered shared for all cores even between different clusters, and non-adaptive— that is, when a page becomes shared it remains shared until it is evicted from the page-table. Further variations of the protocol such as VIPS-H (hierarchical) [63] use more elaborate mechanisms to maintain classification hierarchically across clusters.

Although VIPS-M offers the advantage of simplicity and is employed unmodified within the CPU cluster, its non-adaptive nature prevents us from using it to maintain coherence across devices. VIPS-H is also non-adaptive; furthermore, it requires multi-level classification, and involves multiple interrupts in the transition of a locally-shared page to globally-shared. This prevents us from using it to maintain coherence between devices. Therefore, for
inter-device coherence between the CPU and the GPU we develop VIPS-G, a heterogeneous variation of the VIPS-M protocol that enables adaptive classification between devices, while allowing VIPS-M to operate unmodified within the CPU cluster. Figure 5.2 shows a logical diagram of VIPS-G (right) as opposed to a homogeneous - hierarchical variation of the protocol (left).

The key idea to provide adaptability between devices is to initially lookup if the GPU contains the page before accessing the page-table. This information is volatile, and is available to the CPU through a mirror-TLB (which acts as a shadow-directory). If an entry exists in the mirror-TLB, then the page is shared and therefore, a write-through policy is selected; otherwise the page is considered private and a write-back policy can be safely employed for all the blocks of the page. This allows a page that is shared only between CPU cores to be treated as private by the CPU’s last level cache, for as long as it is not in use by the GPU, become shared when the GPU access it and for as long as it exists in its TLB, and finally revert back to private when it is evicted from the GPU TLB.

5.3 Optimizations for GPGPU computing

GPUs are graphics accelerators with limited general-purpose computing capabilities (GPGPU). Their programming model is more restrictive compared to CPUs, and the hardware has also been designed for graphics, geometrical transformations and fast texture processing. Although these devices have caches and they become increasingly important, some design decisions of the past were not taken with GPGPU application in mind. Therefore, in our work beyond providing UVM, we also try to optimize them for GPGPU computing. State-of-the-art approaches such as [59] and [58], envision GPUs to employ a non-allocate write-invalidate policy for the GPU L1s. Although such a policy

---

1The adaptability between devices, involves two transitions: private to shared and shared to private. The mechanisms to handle these transitions are detailed in Paper IV [41]
might be sufficient for graphics, and further simplify the coherence within the device, it has a huge impact on the performance of GPGPU applications as we show on our work of Paper IV [41]. Therefore, in our approach we employ a write-allocate policy for all GPU caches (L1s and L2).

To maintain coherence within the GPU cluster we use the standard approach of a valid/invalid VI protocol that provides relaxed memory consistency (employed both in commercial GPUs and in academic studies [59, 58, 20, 70]). Therefore, no data classification is used within the GPU. Instead, all valid data are treated as shared and the L1s are flushed at synchronization points. Writing-back all dirty blocks at once during synchronization, however, generates burst-traffic that could enormously increase the synchronization overhead. Therefore, for shared data we use a write-through policy. This has the drawback of generating continuous traffic on writes, which is potentially unnecessary when many writes to the same block may coalesce due to spatial/temporal locality. Therefore, to reduce this traffic we employ small coalescing write-buffers near the caches for the shared blocks.

Finally, for the GPU we propose an energy-efficient memory management unit (MMU) that is built upon virtual coherence and a shared TLB. Driven by the coherence-protocol that does not create any upwards traffic (from L2 to the L1s) we employ virtual coherence for the GPU L1s similar to [35]. This eliminates the need for private TLBs at each GPU core (as opposed to other academic proposals [60]); instead, we use a shared TLB which is accessed before accessing the L2. For as long as we hit in the GPU L1, no translation is required. The translation is required only for accessing the last level (L2) cache of the GPU. Our MMU approach, combines the high-effectiveness of a big shared-TLB with only a fraction of the energy requirements of an MMU with private TLBs. Our final optimization in the system, is to maintain the state of the cache-block (V/I bit) externally in the TLB (similar to [68, 69]) for the CPU-L1s and the GPU-L2. This allows for fast bulk reset of the shared blocks of each page selectively, or for all shared pages.

Figure 5.3 shows the overall design of our system including the TLBs, caches, and address spaces. The CPU is a typical multi-core with few OoO cores having private, discrete instruction- and data-caches, and TLBs; the CPU cores share a big L2 cache. The CPU-L1 caches are virtually-indexed, physically-tagged (VIPT), while the L2 is physically-indexed, physically-tagged (PIPT). Near the CPU-L2 is the mirror-TLB, which is used in inter-device coherence. The GPU is modeled using a custom many-core design. In this design we have a unified instruction-data L1 cache per core and, a shared L2 and TLB across all cores. The GPU L1s are virtually-indexed virtually-tagged (VIVT), while the L2 —similar to the CPU-L2— is PIPT. The page-classification is maintained at the page-table and the TLBs; therefore, those mechanisms also have a role of maintaining coherence in the system (with the help of the OS).
5.4 Evaluation

We evaluate our work using gem5-gpu [59] simulator, which unites gem5 [7] and GPGPU-Sim simulators [5]. Both support detailed simulation mode. The coherency is modeled using the RUBY system as available in gem5. For the power estimates a combination of GPU-Wattch [44] and McPAT [45] is used. As baseline we use the reference implementation of gem5-gpu. This implementation, uses VI-Hammer protocol; a directory-protocol based on MOESI for the CPU and VI for the GPU. Our experimentation shows that the directory is the bottleneck of the system, and severely penalizes the accesses that go through it or modify the coherence state. Related work (Power et al. [58]) also recognizes the problem and proposes a regional coherence approach (using a regional-directory). In our work this bottleneck is inherently removed by removing the directory, while page classification is kept at the TLBs, which is a similar approach to regional directories, except that it has minimal extra overhead.

![Figure 5.3. The memory hierarchy proposed for a fused CPU/GPU system showing the address spaces, the caches, and the TLBs.](image)

![Figure 5.4. Normalized speedup.](image)
We perform our evaluation using a subset of the Rodinia [9] benchmark suite, which has been modified by the gem5-gpu community to work with unified memory address space. Additionally, we include a 7-point stencil kernel from parboil [74] benchmark suite and two applications: wave and octree part, from the workloads used in [70]. Figure 5.4 show the speedup of our approach, compared to a naive directory approach. On average we get \(\approx 45\%\) speedup and \(\approx 45\%\) EDP reduction (\(\approx 20\%\) energy reduction) as shown in Figure 5.5. In some cases —for example, lavaMD and streamcluster the speedup is up to 3x. There are two reasons for the performance improvements (detailed in Paper IV): (i) the GPU-L1 miss-ratio is significantly lower in our approach and, (ii) the access latencies are also significantly lower because our approach eliminates a significant bottleneck: the directory, from the system. Finally, Figure 5.6 shows the detailed energy benefits for the most important system components. For the GPU-core we observe 20\% energy reduction, mostly due to the achieved speedup. For the MMU we observe that our approach requires only a fraction of the energy required by the reference (\(\approx 30\%\)), while both the NoC and cache energy is reduced by on average 20\% and 25\% respectively.

![Normalized EDP and Energy](image1)

**Figure 5.5.** Normalized Total Energy/EDP for VIPS-G (baseline is VI_Hammer).

![Normalized Energy by Component](image2)

**Figure 5.6.** Normalized per component energy for VIPS-G (baseline is VI_Hammer).
6. Related-Work

The decoupled access-execute paradigm was initially proposed by Smith [71] in 1982. Unlike contemporary processors that are widely used today, the execution units of Smith’s DAE processor were unaware of address calculations; instead they were only capable of performing arithmetic operations to the next available operands, provided to them through FIFO queues. In such an approach stalls could occur when the queues became empty. For contemporary processors various approaches have been proposed in the last decade to tackle with memory stalls, such as: helper-threads (e.g., [32, 61, 81]), runahead (e.g., [33]), and slipstream execution (e.g., [27]). The key idea remains the same across all these different approaches: to prefetch data ahead of time, before use by utilizing more resources (either more cores or hardware threads). Employing helper-threads, however, to prefetch data and eliminate stalls in single-threaded applications, provides a potential for performance improvements, with the trade-off of increased energy consumption due to utilizing more cores for the helper threads. DAE substantially differs from these approaches, because the access- and execute-phases run in the same thread.

DAE relies its efficiency on the access-phase, which is based on our ability to generate a lightweight skeleton for the data prefetching. Similar code generation techniques have been used in various inspector-executor approaches [1, 80, 83, 10, 62, 65]. The underlying insight behind those techniques, is to generate an inspector-phase that runs ahead of time and instruments the code, which later on runs optimized at the executor-phase based on the analysis of the inspector. This technique has been widely used to overcome the limitations of static analysis. Other compiler techniques have been proposed to optimize applications’ energy [79, 77, 78, 11, 66, 46] with the use of DVFS. Our approach of Papers II [31] and III [40] combines best of both worlds by employing code transformation to optimize energy within the limits of applicable DVFS of future processors (with low-latency per-core DVFS [38, 37]).

Alternative approaches use the operating system (OS) to profile and select the optimal frequency either for an entire process, or for different phases of a process[36, 72]. Such approaches do not involve code transformation and they typically operate at a coarse granularity defined by the OS time-slice; therefore, they provide a compromise between performance losses and energy savings in codes with high diversity of memory-stall regions.

Software multi-versioning has widely been employed to reduce the overhead of code instrumentation [2, 13, 15, 23], for checking program correctness [19], for loop parallelization, for automatic, speculative optimizations
[12, 30, 49], to optimize the execution for different inputs [75, 82], or even detect races [50]. Our work of Paper III [40] employs software multi-versioning to find the right balance between the prefetching overheads and benefits for the access-phase of DAE programs. Furthermore, it provides a knob to optimize CPU energy efficiency when the CPU runs under different loads.

DAE, also has the potential to improve the energy efficiency of the CPU cores on a heterogeneous architecture, especially when running highly-complex codes as those expected to remain in the CPU even in the heterogeneous era [3], but is not a good fit for the GPU programming model that is capable of tolerating memory stalls. To simplify the programming model, state-of-the-art approaches suggest the use of a unified virtual memory (UVM). For example, the latest versions of NVIDIA CUDA [54] offer UVM. Academic proposals such as HSC [58] and [70] recognize the need for an efficient UVM implementation in hardware. Some approaches like [58] and [59] employ sequential memory consistency models, while other like [70, 20] adopt a release consistency model. Hechtman et al. [21] shows that the consistency model has a minimal impact on performance; however, it has a huge impact on energy consumption and implementation complexity. Therefore, in Paper IV [41] we adopt a release consistency model. Unlike state-of-the-art approaches for SC [60] we minimize the GPU memory management unit overheads by employing virtual coherence similar to [35].
7. Conclusions

This thesis studies efficient execution paradigms for parallel heterogeneous architectures. Our starting point is to improve the energy efficiency of contemporary cores such as those met in many-core and multi-core architectures today, for which we propose the decoupled access-execute (DAE) paradigm. DAE transforms the code in two coarse grain regions: an entirely memory-bound region (called the access-phase) intended to prefetch data into the cache, and another entirely compute-bound (called the execute-phase) that performs the original computation without stalls. The coarse granularity of these phases allows us to set an optimal frequency for each one separately; typically a low frequency is used for the access-phase, while a high frequency is used for the execute-phase. Therefore, DAE allows to optimally DVFS applications for future systems with low-latency, per-core DVFS capabilities.

The decoupled access-execute paradigm has been extensively studied in real hardware, yielding excellent results in different classes of applications. Our experimentation shows great energy savings for predominantly memory-bound applications with irregular control-flow, where the hardware capabilities to hide memory slack are limited. In these applications, DAE significantly improves memory level parallelism (MLP), and therefore, improves both performance and energy efficiency. In our study we evaluate DAE in different application categories such as task parallel and sequential, affine and non-affine, and complex general purpose. Our evaluation shows that DAE has a potential to increase the MLP and the energy efficiency in a wide range of applications.

To facilitate the programmer with an automated way to apply DAE transformations we provide compiler passes to support both affine and non-affine codes. Furthermore, we extend our infrastructure to allow for indirection-measure control of prefetches. We integrate this approach with multi-version DAE (SMVDAE) to allow for dynamic version selection. SMVDAE provides us with a knob to throttle the statically-unknown overhead of the access-phase; therefore, it employs the best access-phase only when it provides benefit over the original (self-adapting and self-healing). Furthermore, SMVDAE has the ability to adapt under different system loads. An SMVDAE system relies on the compiler to statically generate different access versions, and a runtime system to profile and dynamically select the best performing access version.

On average DAE provides excellent results for memory bound applications; not only does it improve energy and EDP but in most of the cases it provides good speedup to memory-bound applications. Without employing multi-versioning, automatic transformations with the help of the compiler
achieved $\approx 20\%$ energy reduction (for the average of the applications evaluated in Papers I and II) and $\approx 20\%$ lower EDP. Employing multi-versioning (as in Paper III) exceeded our expectations by providing $\approx 10\%$ performance improvements and over $30\%$ lower energy for the memory-bound applications, while for the average of the evaluated applications (SPEC-CPU 2006) we achieved over $20\%$ energy savings ($25\%$ EDP).

Finally, in this thesis we explore the possibility to improve the programmer experience on heterogeneous systems. We propose a novel heterogeneous system architecture, which facilitates the programmer with unified virtual memory. Our approach involves minimal modifications (compared to contemporary systems), and introduces minimal overhead to maintain coherence. To define the synchronization scopes we employ the heterogeneous race-free (HRF) memory consistency model. Our goal is to provide regional coherence (at page granularity) with minimal hardware overhead. For that, we use a finely-tuned version of the VIPS coherency protocol (VIPS-G variation), which is asymmetrical and adaptive allowing us to exploit the full potential of heterogeneous system, while at the same time preserving the latency sensitive nature of the CPU cores. Furthermore, we optimize the GPU memory management unit (MMU) by using virtual coherence, and the GPU caches with a more efficient write-allocate policy. Our approach improves performance by $\approx 45\%$ on average and EDP by also $\approx 45\%$. This completes our perspective of a heterogeneous system where DAE is employed to improve the execution paradigm of the sequential part of the application, while the proposed UVM architecture allows for a more efficient execution across the entire heterogeneous system.
Denna avhandling studerar effektiva exekceringsparadigmer för parallella heterogena arkitekturer. De främsta heterogena systemen idag består av en flekärnig CPU och en mångkärnig GPU sammanlagt i samma paket. För CPU är den mest etablerade tekniken för att minska energiförbrukningen dynamisk spänning-frekvensskalning (DVFS). Det mesta av effektiviteten av DVFS förlikar sig på den kvadratiska förhållandet mellan spänning, skalning, och effektförbrukning. I och med slutet av Dennard-skalningen i det senaste årtiondet på grund av den exponentiella ökningen av läckeffekt vid låga spänningar, behövs nya metoder för energihantering. Den underliggande insikten bakom vårt arbete är observationen att prestanda inte alltid är linjärt relaterad till frekvensen (t.ex. i applikationer begränsade av minnes-prestanda) och därför kan vi exploatera denna potential i en programmeringsmodell. Dessutom, för att utnyttja GPU behövs en betydande insats från programmeraren för att kopiera data fram och tillbaka till enheten, samt att synkronisera, och att iscensätta utförandet på ett sådant sätt att dataöverföringar ska överlappas med själva beräkningen. Detta kräver en implementering av ett effektivt enhetligt virtuellt minne (UVM) för hela heterogena system.


I denna avhandling har vi omfattande studerat den frikopplade access - exekveringsparadgimen med riktig hårdvara. Våra experiment visar att DAE ger utmärkta resultat i olika klasser av applikationer. Vi uppnår stora energibespåningar för övervägande minnesbundna applikationer med oregelbundna kontrollflöden, där hårdvarans förmåga att dölja fördröjningar orsakade av datahämtningar är begränsad. I dessa fall förbättrar DAE minnes-parallelлизmen (MLP) avsevärt, och förbättrar därför både prestanda och energieffektivitet. I vår studie utvärderar vi DAE i olika applikationer: uppgifts-parallella eller sekventiella, affina eller icke-affina och komplexa allmänt ändamål; DAE har en potential att öka MLP och energieffektivitet över hela området.

I medel ger DAE utmärkta resultat för minnesbundna applikationer; inte bara förbättras energiförbrukningen och energifördömningsprodukten (EDP), men i de flesta fall ges dessutom en god uppsnabbning till minnesbundna tillämpningar. Utan att utnyttja multi-versionshantering uppnår automatiska omvandlingar med hjälp av kompilatorn omkring 20% minskning av energiförbrukning (genomsnittet av applikationerna som utvärderades i artikel I och II) och omkring 20% lägre EDP. Utnyttjade av multi-versionshantering (som i artikel III) översättrade våra förväntningar genom att tillhandahålla omkring 10% prestandaöverbättringar och över 30% lägre energi för minnesbundna applikationer, medan genomsnittet för de utvärderade applikationer (SPEC-CPU 2006) uppnådde över 20% energibesparingar (25% EDP).

Slutligen, i denna avhandling utforskar vi möjligheten att förbättra programmerarens upplevelse av heterogena system. Vi föreslår en ny heterogen systemarkitektur, vilket förser programmerare med enhetligt virtuellt minne, med minimala modifieringar (jämfört med dagens system) och minimal overhead för att upprätthålla minneskoherens. Vårt tillvägagångssätt utnyttjar den heterogena race-fria (HRF) minnes-konsistens-modellen. Vårt mål är att leverera regional minneskoherens (med en minnes-sida som region) med minimal hårdvaruoverhead. För detta använder vi en finkornig version av VIPS koherensprotokollet (VIPS-G variation), som är asymmetrisk och adaptiv vilket gör det möjligt att utnyttja den fulla potentialen av heterogent system och samtidsigt bevara den latenskänsliga naturen av CPU-kärnor. Dessutom, vi optimerar minneshanteringsenheten (MMU) för GPU, och de privata GPU cachar med virtuellt koherens (vi använder ett VI protokoll inom GPU) och en mer effektiv skriv-allokera politik för GPGPU datoranvändning. Vår ansats förbättrar prestandan med omkring 45% i genomsnitt och även EDP med omkring 45%. Detta fullbordar vårt perspektiv av ett heterogent system där DAE används för att förbättra exekveringsparadigmen för den sekventiella delen av en applikation, medan den föreslagna UVM arkitekturen möjliggör ett mer effektivt utförande över det hela heterogena systemet.
9. Acknowledgments

Firstly, I would like to express my sincere gratitude to my advisor, Prof. Stefanos Kaxiras for his continuous support on this PhD study; for his patience, motivation, and immense knowledge. His guidance was crucial to help me shape into an independent researcher; not only did he taught me how to find interesting research problems, and how to approach them with a novel perspective, but he also taught me how to structure these ideas and convert them into high quality scientific documents. I would also like to express deep gratitude to Prof. Alexandra Jimborean for her overall contribution to the decoupled access-execute paradigm, for bringing her compiler expertise into the project and for all the knowledge transfer regarding LLVM and polyhedral optimizations. Of utter significance to Paper IV and to the work presented in this thesis to support UVM in heterogeneous systems, was the contribution of Prof. Alberto Ros. Therefore, I would like to express my gratitude for all his help in the way to understand the trade-offs in coherence and the endless hours he spent, helping me develop and debug this work. Finally, equal gratitude is expressed to my co-advisors Professors Erik Hagersten and David Black-Schaffer, for the long discussions, for continuously reviewing my work and providing valuable feedback. Working in the UART group has provided me a unique opportunity for me to improve my work, and the way of thinking and presenting it.

But without financial aid, none of this work would be possible. Therefore, I would like to thank all the funders of this work: (i) the EU commission, for offering a 3 year grant under the LPGPU (low-power GPU) project (FP7-ICT-288653), (ii) the Swedish Research Council for co-funding this PhD, and (iii) UPMARC for offering computing resources and support. Finally, I would like to thank and congratulate Uppsala University for offering an employer friendly and motivating environment, and for supporting this work. Great gratitude in this respect is paid to Prof. Ivan Christoff, who as head of the Division of Computer Systems put enormous effort to made the division a better place for the employees, and one of the most highly competitive. Apart from that, he made an example of kindness, politeness, strength and generosity; an example that could shape us into better persons.

Finally, I would like to thank and express my gratitude to all the friends and colleagues for their help and support during those years. More specifically, to Vasileios Spiliopoulos for the long discussions about energy saving trends, and for his effort to build the hardware infrastructure we used to measure power on real machines. I would also like to thank my previous office-mates, Nikos Nikoleris for the long philosophical discussions, and for encouraging me into
the hardest periods; and Muneeb Khan for sharing his experience and expertise on software prefetching techniques, and for all the discussions we had about life in general. As the list becomes longer I would like to thank and express my gratitude to all the master-students that joined the decoupled access-execute project for their contribution. And last but not least, I express endless gratitude to my beloved wife Eirini, for being on my side and withstanding me all those full of deadlines and difficulties years, and for supporting my efforts.
References


[38] Wonyoung Kim, M.S. Gupta, Gu-Yeon Wei, and D. Brooks. System level analysis of fast, per-core DVFS using on-chip switching regulators. In


Acta Universitatis Upsaliensis

Digital Comprehensive Summaries of Uppsala Dissertations from the Faculty of Science and Technology 1405

Editor: The Dean of the Faculty of Science and Technology

A doctoral dissertation from the Faculty of Science and Technology, Uppsala University, is usually a summary of a number of papers. A few copies of the complete dissertation are kept at major Swedish research libraries, while the summary alone is distributed internationally through the series Digital Comprehensive Summaries of Uppsala Dissertations from the Faculty of Science and Technology. (Prior to January, 2005, the series was published under the title “Comprehensive Summaries of Uppsala Dissertations from the Faculty of Science and Technology”.)

Distribution: publications.uu.se
urn:nbn:se:uu:diva-300831