Fix the code. Don't tweak the hardware: A new compiler approach to Voltage-Frequency scaling

Alexandra Jimborean Konstantinos Koukos Vasileios Spiliopoulos
David Black-Schaffer Stefanos Kaxiras

Uppsala University
firstname.lastname@it.uu.se

Abstract

Traditional compiler approaches to optimize power efficiency aim to adjust voltage and frequency at runtime to match the code characteristics to the hardware (e.g., running memory-bound phases at a lower frequency). However, such approaches are constrained by three factors: (i) voltage-frequency transitions are too slow to be applied at instruction granularity, (ii) larger code regions are seldom unequivocally memory- or compute-bound, and, (iii) the available voltage scaling range for future technologies is rapidly shrinking. These factors necessitate new approaches to address power-efficiency at the code-generation level. This paper proposes one such approach to automatically generate power-efficient code using a decoupled access/execute (DAE) model.

In DAE a program is split into tasks, where each task consists of two sufficiently coarse-grained phases to enable effective Dynamic Voltage Frequency Scaling (DVFS): (i) the *access*-phase for data prefetch (heavily memory-bound), and (ii) the *execute*-phase that performs the actual computation (heavily compute-bound). Our contribution is to provide a compiler methodology to automatically generate the access-phases for a task-based programming system. Our approach is capable of handling both affine (through a polyhedral analysis) and non-affine codes (through optimized task skeletons). Our evaluation shows that the automatically generated versions improve EDP by 25% on average compared to a coupled execution, without any performance degradation, and surpasses the EDP savings of the corresponding hand-crafted tasks by 5%.

Categories and Subject Descriptors D.3.4 [*Software*]: Programming Languages, Processors, Compilers; C.0 [*Computer Systems Organization*]: System architectures

Keywords Compiler Optimizations, Polyhedral Model, Task-Based Execution, Decoupled Execution, Performance, Energy, DVFS

1. Introduction

The most widely used technique to reduce power consumption is Dynamic Voltage Frequency Scaling (DVFS) which, by virtue of the power equation[1] $P = aCV^2f$, yields a quadratic power decrease with (at most) linear performance degradation. The common perception is that scaling down voltage and frequency invariably increases power-efficiency, since the quadratic power benefits are bound to outweigh the linear performance losses. This expected DVFS benefit holds if the useful range of voltage scaling (as a function of frequency) is significant enough to make a difference in the total power. Unfortunately, this is no longer the case due to the breakdown of Dennard's scaling in recent process generations[6]. As a result, the useful voltage range is rapidly shrinking to the point of having a negligible effect in the power equation. In practice, we can no longer expect quadratic reductions in power as a trade-off for linear reductions in frequency.

Fortunately, it is also well known that performance is not always proportional to frequency [14]. For example, the performance of memory-bound programs (or program phases) is largely unaffected by frequency scaling. Memory-bound code is characterized by a high miss ratio in the last level cache (LLC), which creates long stalls in the processor pipeline. Scaling down frequency, in this case, causes computation to overlap with memory access without harming the total execution time [13].

Now that voltage scaling is unable to contribute to power savings, it is clear that maximizing opportunities to exploit the non-proportional relationship between frequency scaling and performance is a promising direction. Ideally, we

[1] Where a is the switching activity factor, C is the chip capacitance, V is supply voltage, and f is frequency

would like to scale frequency at the granularity of a cache-missing instruction, i.e., whenever the processor is stalled waiting for memory and only then (otherwise we loose performance without outweighing power benefits). However, frequency transitions are not instantaneous and would impose a steep penalty at this granularity. Previous works focused on identifying memory- and compute- bound regions of code, and setting the voltage and frequency accordingly, thus increasing the granularity [28, 29]. Such region delimitation is performed either based on a fixed interval (execution time), fixed points in the program (branches), guided by profiling, identified by a compiler or manually by the programmer. Such approaches are a compromise, since in most cases computation and memory stalls are intermixed. Scaling down frequency to exploit memory stalls in such coupled code, slows down the coupled computation.

Decoupled access-execute [14] is a task-based approach that modifies programs to have two distinct phases: the *access phase*, which prefetches data into the cache, and the *execute phase*, which does the original computation. The motivation for this partitioning is that the access phase is memory-bound, and can therefore run at low frequencies without a performance loss, while the execute phase will run after the data has been prefetched by the access phase and can therefore run at high frequencies with minimal cache misses. This explicit separation of the code into compute- and memory-bound phases allows better use of DVFS for power saving, due to the coarser granularity of the phases. However, previous work required an expert programmer to manually generate the code for the access phase, limiting the applicability of the method.

Our goal is to *automatically* transform task-based parallel programs into access-execute programs by having the compiler generate the access phase from the original execute phase. Further, a compiler approach has two significant benefits over a manual approach. First, the compiler can derive the access phase *after* applying traditional compiler optimizations to the original (execute) code, thereby leading to leaner access phases; a programmer does not have this option. Second, the compiler is able to apply complex analyses to the memory access patterns and create access phases which are *not* equivalent to the original tasks; performing the same manually is demanding. To automatically generate the access phase at compile time we propose two methods based on statically available information. We demonstrate that these two approaches cover most of our benchmark applications, and result in significant gains in energy efficiency without performance penalties. The first approach uses a polyhedral analysis to examine the memory accesses and generate a new, simplified version, uniquely prefetching the minimal set of addresses touched by the original code. This method is able to generate extremely high-performance access phases, but is restricted to affine codes, due to the limitations of the polyhedral model.

To overcome this limitation, we designed a second method for non-affine codes. Inspired by the *helper threads* and *inspector-execute* techniques, we generate a clone of the original task which contains only the memory accesses and required control flow instructions. By itself, this approach leads to complex access code, which hurts both performance and efficiency. This occurs when the original code contains pointer chasing or complex control flows. To overcome these challenges, we present a set of compile-time optimizations.

Before evaluating this work we first introduce the decoupled access-execute model (Section 3) and explain the power measurement and modeling methodologies used to evaluate its effectiveness (Section 3.2). We then provide an overview of the polyhedral model (Section 4 and Section 5.1), and in particular its limitations and capabilities. From there we walk through our approach to handling codes which are not amenable to the polyhedral model (Section 5.2). Finally, we evaluate the effectiveness of our automatic transformations on a range of benchmarks, comparing them to manually generated access phases.

2. Related work

Power oriented compile-time approaches:

Previous approaches to improve power efficiency have targeted *interval* or *check-point* based DVFS, namely identifying regions that exhibit potential for energy savings by reducing frequency within certain performance degradation bounds [10, 16, 24]. In practice, frequent DVFS switches are inefficient, due to transition overhead and due to the fine granularity of the code regions required to benefit from it, resulting in disappointing power savings or high performance penalties [7]. Hsu et al. [10] propose a compiler algorithm to identify regions in which the CPU is idle due to memory stalls. For these regions, they show that slowing down the processor does not incur significant performance penalties on architectures which allow overlapping of CPU and memory operations, achieving improvements in EDP by 9%, with performance penalties of 2.15% on average.

Heath et al. [8] develop a compiler to perform code transformations to increase processor idle time and inform the operating system about the length of idle periods. By scaling down frequency, this approach reduces energy consumption at the cost of performance. Another frequently adopted strategy known as “race to sleep”, is to highly optimize code for performance such that it completes as quickly as possible. This is well-suited to compute-bound applications, where DVFS does not bring significant benefits [30] and energy savings are just a positive side-effect of performance oriented optimization.

In contrast, Saputra et al. [21] perform loop transformations that were originally intended for performance (tiling, permutation, fusion, distribution), but instead of executing the code at the same frequency to gain performance, they determine a frequency by profiling so that execution time

is unaffected, but energy consumption is reduced. Other approaches [28, 29] apply static techniques complemented by runtime information to perform dynamic compilation to insert DVFS instructions. Such dynamic approaches can adapt better to the inputs or architecture than the purely static methods. For example, the dynamic compilation scheme proposed by Xiang et al. [15] detects hot paths and uses profiling to determine the optimal DVFS, with minimal performance loss.

In contrast to previous approaches, we attempt a better adaptation of the code to the DVFS capabilities, by decoupling the memory accesses of a task from its computation.

Decoupled execution: *Inspector-Executor* techniques [2, 4, 32], traditionally employed in speculative systems, run a skeleton of the code (*inspector*) in advance of the main code. This skeleton performs just the memory accesses to obtain information regarding dependences. The *executor* then follows, and can be optimized using the information obtained from the inspection phase.

In general, such models can be efficient if the address computation is clearly separated from the actual computation, thereby allowing the creation of an efficient inspector. Codes relying on pointer chasing or data-dependent control flow often yield highly complex and inefficient inspectors. To address such problems we have developed optimizations aimed to create a more lightweight access versions which do not degrade performance.

A similar approach, the use of *helper threads*, makes use of a *prefetching thread* to warm up the cache, followed by the worker thread to consume the data, thus hiding memory latencies. Both static [20] and dynamic [31] solutions to generating prefeteching threads have been proposed. Kamruzzaman et al [12] proposed a simplified, profile-guided, manual creation of helper threads, with the prefetching thread and the worker thread running simultaneously. As soon as the worker thread required data already prefetched by the helper thread, it is migrated to the core where the helper thread ran, thus benefiting from the warmed-up cache.

We build upon the decoupled access-execute technique demonstrated by Koukos et al. [14], by automating the creation of an efficient and lightweight decoupled *access* phase out of a task which forms the *execute* phase. Our approach broadens the scope of this technique and allows optimizations and code transformations that are not practical for manual optimization, in many cases producing better results.

3. Background: Decoupled Access-Execute

To evaluate the effectiveness of our automatic compiler-generated access phases, we use a combination of a DAE and DVFS-enabled runtime system and calibrated power and performance models. With these tools we can both accurately measure the power and performance of our generated access phases and predict how effective DAE will be on future systems with more precise DVFS control.

3.1 Matching program behavior to DVFS

Decoupled access-execute is a method of generating coarse-grained phases that expose different program behaviors to take advantage of DVFS. In traditional, *coupled*, execution reducing frequency does help when the processor is stalled waiting for memory, but because the memory and computation are interleaved, it also slows down the computation, resulting in a performance loss. Ideally we would like to run at a low frequency when we are waiting for memory accesses and have no computation to do. Unfortunately, in traditional coupled execution this slack happens at the granularity of just a few instructions, which is far faster than even the fastest DVFS transition overhead on modern processors (with on-chip voltage regulator and with DVFS transitions of nano-scale latencies).

Decoupled access execute (DAE) tries to separate the execution in two coarse-grained phases: a memory-bound access phase (maximal slack) and a compute-bound execute phase (minimal slack). The granularity for the task phases considered in this paper varies from 5-100 usec, making it far more amenable to DVFS.

DAE programs are executed at runtime by having two versions, or phases, of each computation task: the access version that just accesses (prefetches) the data and the execute version that does the original computation. These versions are executed one after another on the same core with the execute immediately following the access phase. The access phase is created by removing the computation that is not needed for address calculation from the execute codes. As an optimization we turn loads into prefetches using the builtin *prefetch* x86 instruction, which does not stall instruction retirement and can therefore provide us with more memory level parallelism (MLP) over simple loads. To ensure no memory faults or incorrect execution paths occur, the compiler generates the access version only when it can be statically proven that: (a) the control flow graph and the computation of memory addresses do not depend on variables visible outside of the task scope (e.g., computing the addresses does not affect any global state), and (b) the task does not contain any function calls which cannot be inlined.

In our framework each task is a well-defined section of code that operates on a small working-set of data. The amount of data a task accesses, its working-set, is a critical parameter for determining the efficiency of the DAE method for two reasons: First, the execute phase should be as compute-bound as possible to maximize performance at max frequency, which means that, ideally, no cache misses should be incurred. This means that the working set of a task should comfortably fit in a core's L1 cache. Second, the DVFS transition latency overhead should be minimized, meaning that we want the *largest* working set that can fit in the L1. As a compromise, we size the task so that its working set just fits the private cache hierarchy of a core (i.e., the L1 and the L2 cache), noting that in an out-of-order core a modest number

of L1 misses that are serviced by the L2 does not affect the "compute-boundedness" of the code and therefore the relationship of performance to frequency.

While the programmer is responsible for selecting the task granularity, the runtime handles task scheduling, running the access phase before the execute phase, load balancing through work stealing and power saving using sleep states and DVFS between each task phase. To choose the voltage and frequency we have two approaches: (a) *naive*, in which we select the lowest frequency for the access phase and the highest for the execute phase, and, (b) *optimal EDP*, in which we try to locally optimize the EDP of each phase by selecting the frequencies that gives us the best EDP for the specific task. (To determine the optimal frequency for a task's EDP we use the power model described in Sec. 3.2.)

Although our framework fully supports DAE-based DVFS and runtime prediction of optimal DVFS for EDP, current systems limit our ability to take full advantage of it in practice. Even in the latest Intel Haswell processors, which feature fast hardware DVFS transitions (with an on-chip voltage regulator), the effective overhead is comparable to that of the previous generation processors (Intel Sandybridge) due to software driver limitations.[1] For this reason our evaluation uses the measured information from our runtime system combined with our power models to predict the benefit that we can achieve when we have low DVFS transition overhead on future systems. To model this we run all the applications at all available frequencies (on the real hardware) and profile the execution time of the access phases, execute phases, and the runtime overhead. Similarly to [14], we measure execution time and model the per-phase power to estimate the overall power and EDP by combining the profiling data from runs across different frequencies. By including the DVFS transition overhead, we can accurately project the effect of our DAE access for different degrees of DVFS latency in future processor generations.

3.2 Power model

For the estimation of energy we employ the power model from [14] in which the processor effective capacitance C_{eff} is expressed as a linear function of the number of instructions executed per cycle (IPC) for the Intel Sandybridge processor. [14] makes use of fine-grained measurements on real hardware to find that $C_{eff} = 0.19 * IPC + 1.64$ and therefore dynamic power is $P_{dynamic} = C_{eff} f V^2$. Static power is modeled as a linear function of voltage-frequency for each number of active cores. The following formula summarizes the total power estimation.

$$P_{total} = \sum_{i=1}^{\#cores} P_{dynamic}(f_i, V_i, IPC_i) + P_{static}(f, V, \#cores)$$

[1] Tested in linux kernels up to version 3.10 using the ACPI driver.

The total energy is $Energy = Time_{total} \times P_{total}$, while the energy delay product $EDP = Time^2_{total} \times P_{total}$, is a more meaningful metric as it takes into account the desire to maintain performance. This power model allows us not only to compute the power estimates for the evaluation of this paper but also to make runtime decisions as to the optimal frequency for each phase.The power model has been evaluated by Koukos et al [14] against real hardware measurements for the whole SPEC 2006 benchmark suite with an average error of 3.1%.

4. Background: the Polyhedral model

The polyhedral model [3, 22] is a powerful mathematical framework for providing a geometric representation of software loops. In the polyhedral model, loop computations and data dependences are represented using integer points in polyhedra. This provides a unified framework for the compiler to reason about complex optimizing transformations, including unrolling, loop-skewing, loop-fusion, loop-fission, parallelization, etc. Polyhedral transformations enable the compiler to perform a reordering of the program's statements, aiming to improve performance and expose parallelization opportunities. Polyhedral optimizations have been shown to have great potential (Pouchet at al. [19] report absolute performance improvements of up to 15×) compared to ICC with automatic parallelization enabled on the original code (Intel ICC 11.0 with options -fast -parallel -openmp).

However, the polyhedral model relies on parametric linear algebra and integer linear programming. This restricts it to loop nests containing static control structures and data accesses which can be represented as affine functions of the enclosing loop indices and parameters (from here on denoted as linear or *affine* codes). This prevents the analysis of code with pointer indirection, data-dependent control flow, or dynamic memory allocation.

The foundation of the polyhedral analysis consists of determining the order and the dependences between the memory accesses. For prefetching purposes, we are only interested in the set of *unique* memory addresses accessed by a task (i.e. considering multiple accesses performed by different instructions to same address only once).

In this work we use the PolyLib [18] library to manipulate the program's polyhedral representation. For *affine* tasks, we rely on the polyhedral abstraction and analysis for building an optimized access version.

5. Compile-time code generation

To automatically generate the access phase, we extend the LLVM compiler [26] to statically generate two versions of the task: (i) the *access* version, designed to prefetch the required data and (ii) the *execute* version, which is the original task with no further modifications. Since creating a lightweight (low-overhead) access phase is crucial for preserving performance, we first turn to the polyhedral model for performing an analysis of the accessed memory locations

and generating an optimized version, as described below, Section 5.1. Since this approach is restricted to affine codes, we handle non-affine codes by building an access version as a skeleton of the original task, containing only instructions required for the computation of the memory addresses to be accessed or in the control flow. For the affine and non-affine approaches we apply different optimization strategies to optimize the performance of the generated access tasks. Code classification is performed at compile time by analyzing the memory locations accessed by each memory instruction. For this purpose, we use LLVM's Scalar Evolution pass, which analyzes loop-oriented expressions and captures how scalars evolve as loops iterate. Based on the expressions provided by the Scalar Evolution pass, we compute linear functions to describe the access pattern of each memory instruction, when possible.

5.1 Affine codes

The generated access code should prefetch only the addresses accessed in the execute phase, and do so as efficiently as possible. However, the two versions need not access *exactly the same* memory locations, nor access them in the *same order*. As the access phase is a *prefetching* operation, correctness is not affected as the actual computation is done by the unmodified execute code. However, the more accurately and quickly the access phase can prefetch the data used by the execute phase, the better the performance and efficiency. This observation is the starting point of a suite of optimizations that result in significant performance gains, compared to an access version built as a simplified clone of the original code. We can therefore generate an entirely new version, whose only role is to prefetch the same memory locations. For affine code, this allows us to fully leverage the abilities of the affine transformation to manipulate the loop accesses to optimize prefetching.

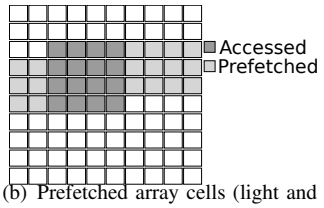
As an example, consider the code in Listing 1(a), which is extracted from our LU benchmark. Figure 1(a) illustrates the memory locations accessed by each instruction. One observes that the set of memory addresses accessed by a 3-depth loop nest represents a matrix, which can be accessed by a loop nest of depth 2. As a result, it is sufficient to generate an access version consisting of a 2-depth loop nest to prefetch these locations more quickly. To achieve this, we perform a static analysis of the accessed memory locations and generate the loop nest of minimal depth prefetching them.

5.1.1 Memory range analysis

A simple approach to determine the set of accessed locations is to compute the memory range accessed by each instruction, parameterized by the loop bounds, and build the union of these ranges. For the example presented in Listing 1(a) accessing the two-dimensional array $A_{N\times N}$, the memory range of addresses accessed by A[i][i] is given by the access

1	4	4	4
2	1,3	3,4	3,4
2	2,3	1,3	3,4
2	2,3	2,3	1,3

(a) Accesses performed by each instruction in the loop nest from Listing 1(a)



(b) Prefetched array cells (light and dark grey) based on the memory range analysis of accessed cells (dark grey)

Figure 1. Memory range analysis: efficient when the whole array is accessed (a), inefficient when accesses target only a block of the array (b)

Listing 1. Code extract from LU:

```
(a) /* Accessing the whole matrix */
for (i = 0; i < N; i++)
  for (j = i+1; j < N; j++) {
    A[j][i](2) /= A[i][i](1);
    for (k = i+1; k < N; k++)
      A[j][k](3) -= A[j][i](2) * A[i][k](4);
  }

(b) /* Accessing a matrix block */
for (i = 0; i < Block; i++)
  for (j = i+1; j < Block; j++) {
    A[j][i] /= A[i][i];
    for (k = i+1; k < Block; k++)
      A[j][k] -= A[j][i]*A[i][k];
  }

(c) /* Prefetch the convex union of
the accessed memory locations */
for (i = 0; i < Block; i++)
  for (j = 0; j < Block; j++)
    prefetch: A[i][j]
```

function[2]: $f(i) = base\_add(A)+(N-1)*i+i, 0 \leq i < N$. This already covers the interval $[base\_add(A),\ base\_add(A)+(N-1)*N]$, which already covers the whole matrix, and is equivalent to the union of all memory ranges. While this solution provides good results when the whole matrix is accessed, as in Listing 1(a), it becomes inefficient for loop nests which only access a block of the matrix, as in the example depicted in Listing 1(b), Figure 1(b). The union of memory ranges in the latter case would return full rows of the matrix, incurring an enormous amount of unnecessary prefetching.

5.1.2 Convex union of accesses

To address this problem, a more fine-grained analysis is required. The solution relies on the polyhedral model to statically detect the exact memory accesses performed by each instruction and to compute the union of these *accesses*, in contrast to computing the union of the *memory ranges*. Once the set of accesses is detected, the compiler generates

[2] Assuming size of an element is 1, for brevity

the loop nest of minimal depth required to prefetch these addresses.

For detecting the exact set of *all* accessed memory locations, we rely on the polyhedral framework to compute the set of addresses accessed by each instruction and the union of these sets (one set per instruction). In order to generate the simplest and most efficient loop nest scanning the union of memory addresses, we compute the *convex hull of the union*.

For the example in Listing 1(b), one obtains the convex union of the accessed addressed, as a polyhedron:

$$\{\text{i,j} \mid 0 \leq \text{i} \leq \text{Block}, 0 \leq \text{j} \leq \text{Block}\}.$$

From this representation, the compiler automatically generates the access phase consisting of the loop nest in Listing 1(c).

Trade-offs and solutions

1. Wide convex hull: The trade-off of computing the *convex hull of the union of accessed memory addresses* is that it may be too large. By definition, to ensure convexity, the convex hull includes the memory areas (whether accessed or not) between the accessed locations. Therefore, as in the case of the memory range analysis, it may also prefetch a number of unnecessary locations. We propose a simple method to ensure that the generated loop nest for prefetching the memory locations does not scan any un-accessed addresses. The solution we propose is to count the number of memory locations accessed in the original loop nest ($NOrig$), and the number of locations contained in the convex union ($NconvUn$). In our approach, we decide to generate the loop nest scanning the convex hull only if: $NconvUn \leq NOrig$. Nevertheless, one can design heuristics to determine a threshold th, $NconvUn - th \leq NOrig$, such that this optimization still provides benefits. This is equivalent to computing the number of *unnecessarily prefetched memory locations* allowed, without hurting performance.

To compute $NOrig$, we need to determine the exact set of locations accessed by each instruction and their (non-disjoint) union, represented as a union of $\mathbb{Z}$-polytopes [23]. Next, we sum up the number of integer points contained in each of these polytopes, using Ehrhart polynomials [5]. Similarly, $NconvUn$ is computed by counting the integer points in the convex union.

2. Loop nests accessing different arrays: For loop nests that access multiple arrays (such as in Listing 2(a) which accesses arrays A and D), the access version should efficiently prefetch data from each array. Ideally, this requires generating a unique loop nest that can scan both arrays, as depicted in Listing 2(b), to minimize the overhead.

To achieve this, we compute the convex hulls corresponding to the set of accesses to each array and generate the corresponding loop nests. Next, we merge these loop nests into one, only if they have the same number of iterations. As before, we could consider relaxing this constraint and merge the loops if the number of iterations differ by a certain threshold if this improves performance.

Listing 2. 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]
  }
```

3. Loop nests accessing different blocks of the same array: A particularly interesting situation is when a loop nest accesses different blocks of the same multi-dimensional array, as in Listing 3.

Listing 3. 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]
  }
```

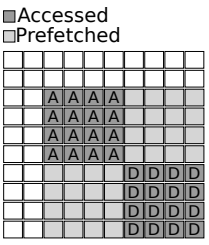


Figure 2. Accesses to blocks of a matrix: classA, classD

In such cases the convex hull would include the accessed blocks, together with the memory areas in-between as depicted by the light grey area in Figure 2. Our solution to avoid unnecessary prefetching in this case is to separate memory accesses into classes which use the same parameters. In the example in Listing 3(a), we would have $classA$ (depending on parameters Ax and Ay) and $classD$ (depending on Dx, Dy) as shown in Figure 2. Next, as in the example with accesses to different arrays, we compute the loop nests individually, and we merge them if the number of iterations coincide. Hence, we obtain the access version illustrated in Listing 3(b), efficiently prefetching only the locations illustrated as dark grey cells in Figure 2.

5.2 Access version for non-affine codes

For code that is not amenable to the polyhedral model, the compiler starts out by creating a simplified clone of the original loop nest. However, this simple approach generates inefficient access codes, and needs to be complemented with several optimizations to produce acceptable results.

5.2.1 Straightforward approach and its refinements

Intuitively, the access code generated from the execute code should retain instructions that determine the control flow and compute memory addresses, and should replace *load* and *store* instructions (read/write accesses in the LLVM intermediate representation) with *prefetch* instructions. This leads to a naive implementation that simply prunes instructions not needed for memory access and replaces loads and stores with prefetches. Empirically, we discovered that prefetching the memory addresses accessed for writing does not improve performance, hence, we discard the *store* instructions and only prefetch the read memory accesses[3]. To improve on this naive approach, we:

- prefetch only global variables and values transmitted as parameters to the task;
- detect the set of accessed addresses and prefetch each of them only once;
- *accompany*, rather than replace, each *load* with a prefetch instruction, relying on dead code elimination to remove instructions that are not required for memory address computation nor for preserving the control flow graph.

Finally, the generated access code (denoted access version) is optimized using traditional compile time optimizations (-O3).

5.2.2 Simplified CFG

One drawback of creating simplified versions using the previous approach to prefetch the accessed memory addresses is that the access versions might actually be as complex as the original execute code. Thus, not only the address calculation but also part of the computation is replicated in the access version, turning it into a heavy code which is no longer only memory-bound. Such situations commonly occur when the code has a data-dependent control flow.

To avoid such cases and to limit the overhead, we perform a simplification of the control flow graph in the access version, by eliminating conditionals embedded in loop bodies, if they do not maintain the control flow of the loop. This optimization has two consequences. First, the access phase is considerably faster, since it contains only a simple control flow. Second, by eliminating the conditionals, we ensure that only data which is guaranteed to be accessed in all iterations is prefetched, thus reducing unnecessary prefetching.

[3] Part of this is due to the less critical nature of store instructions as they are unlikely to stall the processor pipeline during the execute phase.

Algorithm summary: The algorithm first marks all instructions required to generate the prefetch addresses and then discards the remaining instructions when generating the access code. The main steps are listed below:

1. *Inline function calls in the task, when possible.* If any function calls cannot be inlined, we do not generate an access version to avoid unwanted side-effects.
2. *Create an identical clone of the task*. By creating a copy, all local variables of the original task are *privatized* in the clone access version.
3. Identify and *mark* uses (reads) of variables visible outside the scope of the task (global variables, function arguments) and associate corresponding prefetch instructions.
4. Identify and *mark* instructions required to preserve the loop control flow.
5. Starting from the marked instructions (reads and loop control flow), identify and *mark* address computations and values required for the loop control flow by following the use-def chain. If a write to a value visible outside the task boundaries is required either for computing memory locations or for maintaining the loop control flow, then we do not generate an access version.
6. Finally, *discard* all *unmarked* instructions. Followed by dead code elimination, this step removes unnecessary computations and branches. To further simplify the control flow, reads of variables visible outside the task, which are not guaranteed to execute (e.g. embedded in conditionals) are also discarded. Note however, that instructions required for the control flow of the loop (such as instructions determining the loop exit) must be preserved whether they execute conditionally or not.

While eliminating conditionals within loops gives a general improvement, some applications would benefit from the additional or more precise prefetching of keeping the conditionals. This is likely if particular conditional-branches are executed for the majority of the iterations. To address such situations, we could detect the hot path through profiling and create a specifically tailored access version. Moreover, optimization opportunities in codes that exhibit phases could be explored by means of multiple statically generated access versions selected based on the appropriate phase at runtime.

5.2.3 Avenue of further optimizations

There are numerous optimizations that could improve the efficiency of the decoupled access-execute model, including: prefetching only one access per cache line, instead of per memory address; avoiding recomputation of memory addresses; and adjusting the granularity of the task automatically at compile-time to optimize the amount of data prefetched by the access phase.

6. Evaluation

We evaluated our automatic access phase generation on a selection of benchmarks ranging from compute- to memory-bound applications: *LU, Cholesky* and *FFT* (SPLASH2 [27]) are examples of computational intensive kernels, *CIGAR* [1] and *libquantum* (SPEC CPU2006 [9]) are memory-bound, while *CG* (NAS parallel benchmark suite [17]) and *LBM* (SPEC CPU 2006 [9]) exhibit an intermediate behavior. These applications were manually ported to a task-based runtime where the granularity of each task can be adjusted by a runtime parameter. Energy and performance numbers were generated using the runtime and power models described in Section 3.2, following the configuration displayed in Table 1.

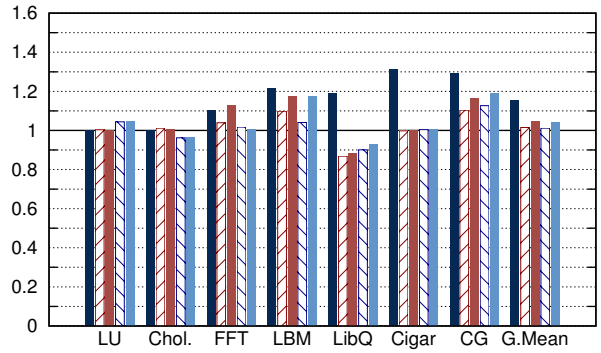
Table 1. Application characteristics. # affine loops: number of loops handled with the polyhedral approach, out of the total number of target loops; $T_A\%$: Average fraction of the application's execution time spent in the access phase; $T_A(usec)$: Average duration of the access phase.

Application	# affine loops / total loops	# tasks	T_A %	T_A (usec)
LU	3/3	89440	1.83	6.82
Cholesky	3/3	45760	1.80	6.05
FFT	0/6	82304	19.24	30.74
LBM	0/1	2600192	47.95	7.90
LibQ	0/6	51603486	47.01	2.64
Cigar	0/1	10576778	49.27	5.11
CG	0/2	35634375	42.84	2.89

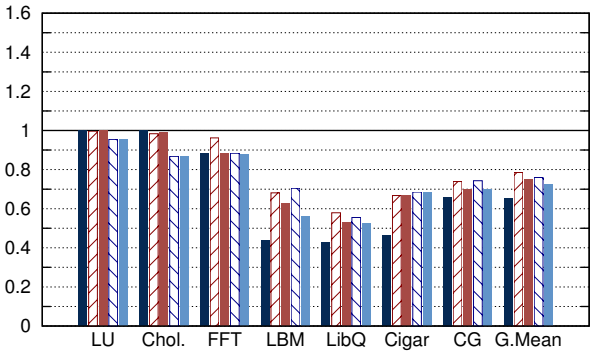
6.1 DAE *vs.* regular task execution

We compare the results of DAE execution in terms of time (performance), energy and EDP vs.: (1) CAE: original task execution (coupled access-execute), (2) Manual DAE: decoupled access-execute wherein the access version was manually crafted by an expert programmer, and, (3) Auto DAE: decoupled access-execute wherein the access version was automatically generated by the compiler. In Figure 3, results are normalized to the original task coupled execution (CAE) running at the Linux default maximum frequency. $Min/max\ f$ data is from running the access phase at the lowest frequency and the execute phase at highest frequency. $Optimal f$ selects the most suitable frequency for each phase of an application in order to achieve optimal EDP, using a prediction model which relies on offline profiling, as discussed in Section 3.2.

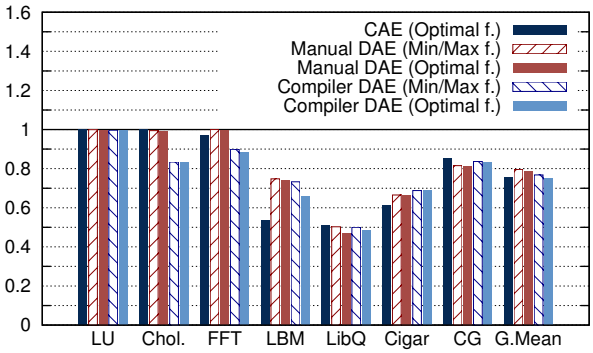
Since the focus of this work is to demonstrate the potential of DAE, we perform an exhaustive search to select the optimal frequency in terms of EDP. Online techniques, however, have also proven to be accurate enough to minimize EDP by scaling frequency at runtime. It has been demonstrated that analytical-based models can be approximated using performance counter events on real hardware to achieve near-optimal EDP savings [11, 25]. For future hardware with lower DVFS transition latencies, similar approaches will be effective for optimal DEA frequency selection.



(a) Time (Normalized to Max Frequency)



(b) Energy (Normalized to Max Frequency)



(c) EDP (Normalized to Max Frequency)

Figure 3. Performance, Energy, and EDP results on a quad-core Intel Sandybridge, assuming state-of-the-art 500ns frequency scaling transition latency.

DEA can significantly improve the Energy × Delay Product (EDP) with negligible impact on performance, as demonstrated by Koukos et al. [14]. While both coupled and decoupled executions benefit from DVFS, coupled execution shows a significant performance penalty when the frequency is reduced to save energy. In contrast, the decoupled execution preserves performance in addition to reducing the energy consumption. Note that decoupled execution delivers benefits for both compute- and memory-bound applications, and provides a similar EDP improvement to that of the lower-frequency coupled execution. An exception to this overall trend is the *LBM* benchmark, in which the EDP im-

provement for coupled execution is larger than for decoupled, due to the fact that its *write accesses* are coupled with computations during the execute phase. Thus, this application does not take the complete advantage of the decoupled execution which decouples only the *read accesses* from computation.

Our experiments were carried out assuming the frequency scaling transition latency of current generation processors (Intel Haswell), estimated to 500 nanoseconds. During each DVFS transition we count only the static energy, since no instructions are executed. We also evaluated our approach considering the ideal case of instant per-core DVFS of future processors. Assuming zero transition latency, both Manual DAE and Auto DAE slightly outperform CAE at max frequency, regarding execution time. Moreover, Manual DAE yields 25% EDP improvement with $Optimal f$ policy, while Auto DAE delivers 29% EDP improvement (geometric mean across all applications). Considering the more realistic transition latency of 500ns, both DAE approaches pay a performance penalty of approximately 4% with $Optimal f$ policy, while improving EDP by 23% (Manual DAE) and 25% (Auto DAE) respectively, as shown in Figure 3.

An in depth analysis comparing the decoupled and coupled execution from performance and energy viewpoints is presented in previous work by Koukos et al [14].

6.2 Manual DAE *vs.* Auto DAE

To evaluate the effectiveness of the compiler optimizations introduced with this work we examine detailed time and energy results across coupled execution (CAE), Manual DAE, and Auto DAE in Figure 4. We chose as case studies 3 applications where the Auto DAE and the Manual DAE versions differ in performance and energy consumption: (a) *Cholesky*, where the compiler generates the access version guided by polyhedral analysis, (b) *FFT* and (c) *LibQ*, for which the automatically built access versions are optimized skeletons of the original tasks. Each graph displays the behavior of CAE, Manual DAE and Auto DAE as a function of frequency, left-to-right from f_{min} (1.6GHz) to f_{max} (3.4GHz), in steps of 400MHz. For both DAE versions, the access phase is executed at f_{min}, while the execute phase is varied from f_{min} to f_{max}.

6.2.1 Cholesky

Cholesky is a compute-bound kernel. Consequently, a straightforward generation of an access version preserving the memory accesses would incur a performance degradation of up to 1.7× compared to the original execution time, with significant energy penalties as a consequence. However, since Cholesky is an affine kernel, it can be abstracted to a polyhedral representation, thus enabling advanced analysis of the accessed memory locations and the generation of a highly optimized and more efficient access version.

The manually created version is similarly optimized, but performs selective prefetching, thus less data is actually brought in the cache. As a result, the access phase has a shorter execution time compared to the automatically generated one, but the overall execution time is slightly greater because the execute phase has to fetch the missing data. From an energy viewpoint, the automatically generated access version outperforms the hand-crafted one because it saves more energy by prefetching more data at the lower frequency and then executing for less time at the higher frequency. This example demonstrates that while the automatically generated access phase prefetches more data than the manually written one, the tradeoff is a win in energy.

6.2.2 FFT

The parallel tasks of the *FFT* kernel contain calls to other functions, each of which contain a number of loop nests. Compile time optimizations inline these functions and perform advanced loop optimizations, merging the loop nests originating from different functions. Thus, a simplified and more efficient version of the original task is available to the compiler as a starting point in building the access version. Conversely, the manually crafted version was generated from the unoptimized source code. Yet, the hand-made version uses the expert's knowledge regarding data accesses and is greatly simplified. By being simpler, the manual access version completes faster than the auto-generated version, but prefetches less data, hence the execute phase requires longer time to complete. From the performance standpoint, both Manual DAE and Auto DAE are competitive with the original coupled execution. Nevertheless, the advantage of the automatic version consists in an access phase prefetching more data while running at a low frequency, which reduces the energy consumption and yields overall improvements in EDP.

6.2.3 LibQ

LibQ is an example where both Manual DAE and Auto DAE access versions were generated as an optimized clone of the original task. Additionally, Manual DAE eliminates redundant prefetch instructions (i.e. targeting data residing in the same cache line, such as different fields of a complex data structure). Such optimizations are possible for the expert having knowledge of the data layout, but difficult to discover at compile-time. A solution to overcome this limit and improve the automatically generated access versions is to perform a profiling step to identify instructions that regularly incur cache misses and only prefetch those.

The effect of selective prefetching in *LibQ* is a faster manually crafted access phase, compared to the automatically generated version. Although the execute phase in Auto DAE requires shorter time to complete, the total execution time is slightly increased. On the other hand, the benefit of a longer executing access version translates in energy gains, hence both Manual DAE and Auto DAE yield similar EDP.

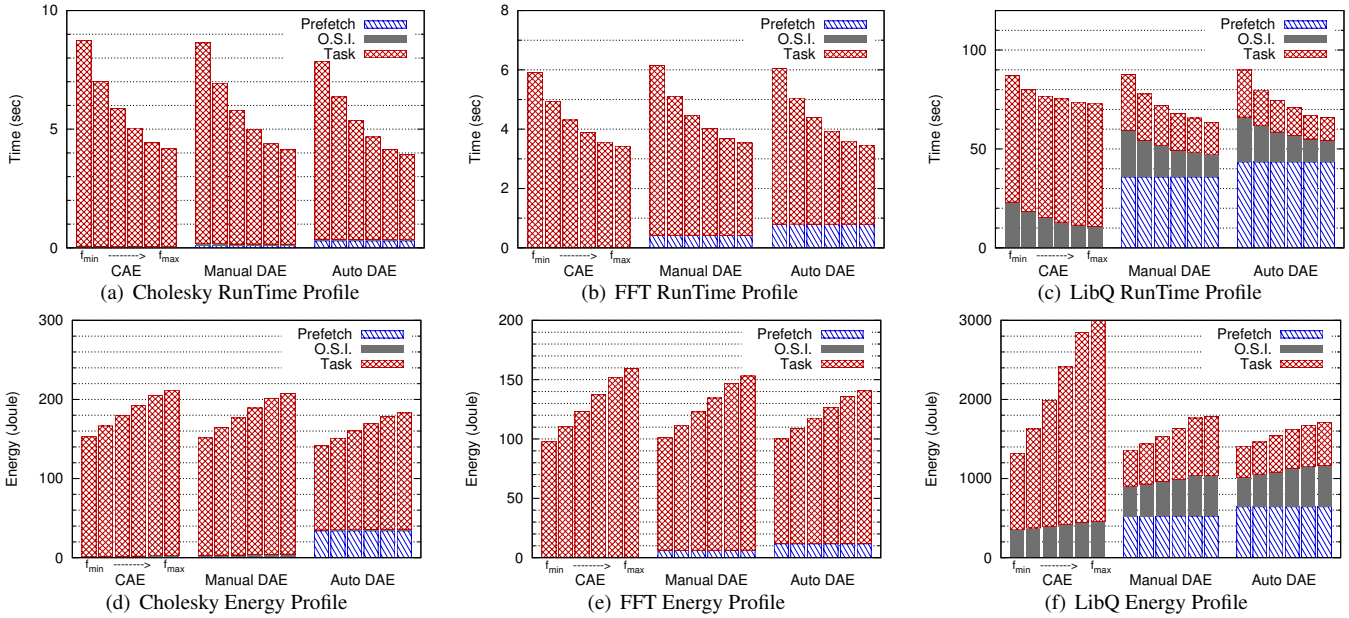


Figure 4. Comparison of coupled execution, Manual DAE and Auto DAE as a function of execute frequency. O.S.I. is the aggregate of **O**verhead, **S**equential, and **I**dle times (typically includes all time that is not within task access or execute).

6.2.4 Conclusions

In two of these examples (*Cholesky* and *FFT*) we see that while the Auto DAE access code takes longer to execute than the Manual DAE access code, it does not result in an overall increased execution time, whereas in *LibQ*, the performance advantage gained by the Manual DAE code is minimal. This means that the overall proportion of execution time in the access phase is increased, which allows the runtime to run at the lowest frequency for a greater portion of the time, resulting in an overall improvement in EDP compared to the Manual DAE code. The three benchmarks analyzed in detail reflect the results obtained in all other evaluated applications.

7. Conclusions

In this work we have demonstrated new compiler optimizations to improve energy efficiency via automatic access phase generation for decouple access execute. To exploit the full potential of DVFS, we perform code transformations that generate a memory-bound *access* version of each task. This allows the DAE runtime to execute the access task immediately before the execute task, which prefetches its data and makes the original execute task effectively compute-bound, since the data is already available in the cache. Having coarse-grained memory- and compute-bound execution phases allows the runtime to adjust frequency accordingly, exploiting the maximum potential of DVFS.

Unlike previous approaches relying on a decoupled access-execute model (as for instance the inspector-executor paradigm used in dynamic speculative parallelization) we do not require *equivalent* access and execute phases because our access phase is a speculative prefetch. This enables us to develop advanced static analyses to generate highly optimized, although not equivalent, access versions. As a result we can preserve or even enhance overall performance by prefetching data with a minimal overhead. For memory bound applications we have a significant EDP improvement of up to 50% and 25% on average compared to coupled execution.

Using these optimizations, we compare the automatically generated access code with hand-crafted versions and demonstrate that the compiler generated versions are competitive both in performance and energy efficiency. Moreover, these complex static optimizations even exceed the ones prepared by an expert in several benchmarks. As future work, we consider employing a profiling step in guiding static transformations, by complementing the information retrieved at compile time.

Acknowledgements

We thank Vincent Loechner for his insights on the polyhedral model and his valuable help. This work is supported, in part, by the Swedish Research Council UPMARC Linnaeus Centre and the EU Project: LPGPU FP7-ICT-288653.

References

[1] Cigar - case injected genetic algortihm. URL `http://www.cse.unr.edu/~sushil/class/gas/code/cigar/`. http://ecsl.cse.unr.edu/.

[2] M. Arenaz, J. Tourio, and R. Doallo. An inspector-executor algorithm for irregular assignment parallelization. In *2nd Int'l Symp. on Parallel and Distributed Processing and Applications (ISPA)*, pages 4–15, 2004.

[3] U. K. Banerjee. *Loop Transformations for Restructuring Compilers: The Foundations*. Kluwer Academic Publishers, 1993.

[4] D. K. Chen, J. Torrellas, and P. C. Yew. An efficient algorithm for the run-time parallelization of doacross loops. In *ACM/IEEE Conf. on Supercomputing*, pages 518–527, 1994.

[5] P. Clauss. Counting solutions to linear and nonlinear constraints through ehrhart polynomials: applications to analyze and transform scientific programs. In *10th Int'l Conf. on Supercomputing*, pages 278–285, 1996.

[6] R. Dennard, F. Gaensslen, V. Rideout, E. Bassous, and A. LeBlanc. Design of ion-implanted mosfet's with very small physical dimensions. *Solid-State Circuits, IEEE Journal of*, 9 (5):256 – 268, 1974.

[7] D. Grunwald, C. B. Morrey, III, P. Levis, M. Neufeld, and K. I. Farkas. Policies for dynamic clock scheduling. In *4th Conf. on Symp. on Operating System Design & Implementation - Volume 4*, 2000.

[8] T. Heath, E. Pinheiro, J. Hom, U. Kremer, and R. Bianchini. Application transformations for energy and performance-aware device management. In *Parallel Architectures and Compilation Techniques*, pages 121–130, 2002.

[9] J. L. Henning. Spec cpu2006 benchmark descriptions. *SIGARCH Comput. Archit. News*, 34(4):1–17, 2006. URL `http://doi.acm.org/10.1145/1186736.1186737`.

[10] C.-H. Hsu and U. Kremer. The design, implementation, and evaluation of a compiler algorithm for cpu energy reduction. *SIGPLAN Not.*, pages 38–48, 2003.

[11] C. Isci, G. Contreras, and M. Martonosi. Live, runtime phase monitoring and prediction on real systems with application to dynamic power management. In *Int. Symposium on Microarchitecture*, 2006.

[12] M. Kamruzzaman, S. Swanson, and D. M. Tullsen. Inter-core prefetching for multicore processors using migrating helper threads. *SIGPLAN Not.*, pages 393–404, 2011.

[13] G. Keramidas, V. Spiliopoulos, and S. Kaxiras. Interval-based models for run-time dvfs orchestration in superscalar processors. In *7th ACM Int'l Conf. on Computing frontiers*, pages 287–296. ACM, 2010.

[14] K. Koukos, D. Black-schaffer, V. Spiliopoulos, and S. Kaxiras. Towards More Efficient Execution : A Decoupled Access-Execute Approach Categories and Subject Descriptors. In *ICS'13*, 2013.

[15] X. LingXiang, H. JiangWei, S. Weihua, and C. TianZhou. The design and implementation of the dvs based dynamic compiler for power reduction. In *7th Int'l Conf. on Advanced parallel processing technologies*, pages 233–240, 2007.

[16] J. R. Lorch and A. J. Smith. Improving dynamic voltage scaling algorithms with pace. *SIGMETRICS Perform. Eval. Rev.*, 2001.

[17] NASA. Nas parallel benchmarks, 1999. URL `http://www.nas.nasa.gov/assets/pdf/techreports/1999/nas-99-011.pdf`.

[18] PolyLib. A library of polyhedral functions. `http://icps.u-strasbg.fr/polylib/`.

[19] L.-N. Pouchet, U. Bondhugula, C. Bastoul, A. Cohen, J. Ramanujam, P. Sadayappan, and N. Vasilache. Loop transformations: convexity, pruning and optimization. In *38thACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages*, pages 549–562, 2011.

[20] C. G. Quiones, C. Madriles, J. Snchez, P. Marcuello, A. Gonzlez, and D. M. Tullsen. Mitosis compiler: An infrastructure for speculative threading based on pre-computation slices. In *ACM SIGPLAN Conf. on Programming language design and implementation*, pages 269–279, 2005.

[21] H. Saputra, M. Kandemir, N. Vijaykrishnan, M. J. Irwin, J. S. Hu, C.-H. Hsu, and U. Kremer. Energy-conscious compilation based on voltage scaling. In *Joint Conf. on Languages, compilers and tools for embedded systems: software and compilers for embedded systems*, pages 2–11, 2002.

[22] A. Schrijver. *Theory of linear and integer programming*. John Wiley & Sons, Inc., NY, USA, 1986. ISBN 0-471-90854-1.

[23] R. Seghir and V. Loechner. Zpolytrans: a library for computing and enumerating integer transformations of z-polyhedra. In *2nd Int'l Workshop on Polyhedral Compilation Techniques (IMPACT'12)*, 2012.

[24] T. Simunic, L. Benini, A. Acquaviva, P. Glynn, and G. De Micheli. Dynamic voltage scaling and power management for portable systems. In *Design Automation Conf.*, pages 524–529, 2001.

[25] V. Spiliopoulos, S. Kaxiras, and G. Keramidas. Green governors: A framework for continuously adaptive dvfs. In *Int'l Green Computing Conf. and Workshops*, IGCC '11, pages 1–8, 2011.

[26] The LLVM. Compiler infrastructure. `http://llvm.org`.

[27] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The splash-2 programs: characterization and methodological considerations. In *22nd Annual international Symp. on Computer architecture*, pages 24–36, 1995.

[28] Q. Wu, M. Martonosi, D. W. Clark, V. J. Reddi, D. Connors, Y. Wu, J. Lee, and D. Brooks. A dynamic compilation framework for controlling microprocessor energy and performance. In *38th Annual IEEE/ACM Int'l Symp. on Microarchitecture*, pages 271–282, 2005.

[29] Q. Wu, M. Martonosi, D. W. Clark, V. J. Reddi, D. Connors, Y. Wu, J. Lee, and D. Brooks. Dynamic-compiler-driven control for microprocessor energy and performance. *IEEE Micro*, pages 119–129, 2006.

[30] T. Yuki and S. Rajopadhye. Folklore confirmed: Compiling for speed = compiling for energy. In *26th Int'l Workshop on Languages and Compilers for Parallel Computing*, 2013.

[31] W. Zhang, D. M. Tullsen, and B. Calder. Accelerating and adapting precomputation threads for effcient prefetching. In *IEEE 13th Int'l Symp. on High Performance Computer Architecture*, pages 85–95, 2007.

[32] C.-Q. Zhu and P.-C. Yew. A scheme to enforce data dependence on large multiprocessor systems. *IEEE Trans. Softw. Eng.*, 13, 1987.