Multiversioned Decoupled Access-Execute: The Key to Energy-Efficient Compilation of General-Purpose Programs

Konstantinos Koukos
Uppsala University, Sweden
konstantinos.koukos@it.uu.se

Per Ekemark
Uppsala University, Sweden
Per.Ekemark.8846@student.uu.se

Georgios Zacharopoulos
Switzerland Università della Svizzera italiana, Switzerland
georgios.zacharopoulos@usi.ch

Vasileios Spiliopoulos
Uppsala University, Sweden
vasileios.spiliopoulos@it.uu.se

Stefanos Kaxiras
Uppsala University, Sweden
Stefanos.kaxiras@it.uu.se

Alexandra Jimborean
Uppsala University, Sweden
alexandra.jimborean@it.uu.se

Abstract

Computer architecture design faces an era of great challenges in an attempt to simultaneously improve performance and energy efficiency. Previous hardware techniques for energy management become severely limited, and thus, compilers play an essential role in matching the software to the more restricted hardware capabilities. One promising approach is software decoupled access-execute (DAE), in which the compiler transforms the code into coarse-grain phases that are well-matched to the Dynamic Voltage and Frequency Scaling (DVFS) capabilities of the hardware. While this method is proved efficient for statically analyzable codes, general-purpose applications pose significant challenges due to pointer aliasing, complex control flow and unknown runtime events. We propose a universal compile-time method to decouple general-purpose applications, using simple but efficient heuristics. Our solutions overcome the challenges of complex code and show that automatic decoupled execution significantly reduces the energy expenditure of irregular or memory-bound applications and even yields slight performance boosts. Overall, our technique achieves over 20% on average energy-delay-product (EDP) improvements (energy over 15% and performance over 5%) across 14 benchmarks from SPEC CPU 2006 and Parboil benchmark suites, with peak EDP improvements surpassing 70%.

Categories and Subject Descriptors D.3.4 [*Software*]: Programming Languages, Processors, Compilers

Keywords Decoupled access-execute, Energy efficiency, DVFS, Compile-time transformations, Multi-versioning

1. Introduction

Energy has become the limiting factor in computer systems, with power and cooling cost limiting server performance, and battery runtime limiting mobile and embedded systems performance. One of the primary tools for improving power efficiency of applications has been the use of dynamic voltage and frequency scaling (DVFS). DVFS relies on the quadratic relationship between voltage and power, and the linear relationship between voltage - frequency and performance. This provides quadratic power savings with at most linear performance loss and, has allowed processors to achieve lower power consumption with only a small penalty in performance for the past decade.

However, the effectiveness of DVFS has been significantly hurt by the increasingly small voltage ranges of modern silicon transistors as the supply voltage approaches the threshold voltage (known as the end-of Dennard scaling [9]). As the effect of voltage scaling diminishes, we are left only with the ability to reduce frequency to control processor power. While reducing voltage used to provide quadratic energy savings for at most linear performance degradation, frequency scaling alone only provides linear energy savings for a linear performance degradation.

To continue reducing energy while maintaining performance, a promising opportunity exists in exploiting the performance gap between the processor and memory system [11, 25, 31, 38]. Ideally, frequency would be scaled down while waiting for data to be fetched from memory, *i.e.* upon a cache miss, and scaled up while performing computations. Nevertheless, adjusting frequency at such a fine granularity is not possible on current processors and furthermore, such frequent changes would incur prohibitive overheads. To adapt to this new reality, recent work has demonstrated that compilers have become the modern means for further providing high performance at low energy, by adjusting software to match DVFS capabilities.

Background: A key example is the software decoupled access-execute (DAE) technique [21, 26] demonstrated on task-based parallel programs. This approach splits task execution into coarse-grained *memory bound-phases*, executed at low frequency to save energy since the processor is waiting for data and does not benefit from a higher frequency, and *compute-bound phases*, executed at high frequency to preserve performance. As a result, DAE is able to achieve significant energy savings with negligible impact on performance.

CC'16, March 17–18, 2016, Barcelona, Spain
© 2016 ACM. 978-1-4503-4241-4/16/03...$15.00
http://dx.doi.org/10.1145/2892208.2892209

Splitting applications in memory-bound and compute-bound phases with coarse granularity is challenging. To address this, a compiler pass was designed to automatically generate access and execute phases for each task in task-based parallel programs [21]. The compiler built a memory-bound version derived from the original task, called the *access phase*, designed to prefetch the data in the cache, and used the original task as the *execute phase*. This approach demonstrated a 25% energy savings without hurting performance.

However, the compiler techniques designed for scientific applications exploited the benefits of statically analyzable, affine code to model the memory accessing behavior of a task in the polyhedral model. Sequential, general-purpose applications rarely exhibit affine behavior, thus they are not amenable to a polyhedral representation. Non-affine codes were handled in DAE [21] with heuristics for balancing the effectiveness and the overhead of the access phase. More precisely, to simplify the control flow, DAE prefetches only accesses which are not guarded by conditionals. Moreover, DAE prefetches the last indirection of memory accesses performed through indirections. Applying these heuristics on general purpose code with highly complex control flow would lead to a highly inefficient access phase, since the delinquent loads often reside within conditionals. On the other hand, duplicating the entire control flow in the access phase would introduce a prohibitive overhead. Similarly, prefetching the last indirection has the potential to generate heavy access phases, since in general purpose codes it is common to encounter up to 20 indirections. This could significantly slowdown the application.

Contributions: To overcome these limitations, we introduce a new approach to automatically transform legacy sequential codes at compile time for enabling energy-efficient execution via decoupled access execute. General-purpose code abounds in statically unknown events, calling for new compiler techniques to deliver the right balance between lightweight and efficient access phases. Even in case DAE is successful in generating access phases for the complex general-purpose code, it is impossible to determine statically the effect on performance. **The compiler must consider the trade-off between the overhead of computing the address to prefetch (which may depend on other memory accesses or conditionals, cache content, input data, etc.) and the benefits obtained from prefetching the data.** These issues are addressed through a simple, but efficient, software *multi-versioning* strategy (SMVDAE), which is build upon *statically* generating a number of access versions ranging from very lightweight (and potentially less effective at prefetching the data into the cache) to more complex (and potentially more effective, but with higher overhead). The best performing access phase is selected dynamically, thus overcoming the limitations of static analysis. The advantage is twofold: (i) SMVDAE can explore the search space of optimized access versions at runtime, and (ii) it achieves this efficiently —that is, SMVDAE incurs no runtime overhead in generating optimized code, since multi-versioning is done statically, and adds only a minimal overhead for selecting the optimal version dynamically.

One characteristic of DAE, and implicitly of SMVDAE as a successor, is that access phases are side-effect free, namely, no writes to variables that escape the scope of the access phase are allowed. Yet, in some situations, to generate the access phase, a number of stores are required, whose addresses cannot be always statically disambiguated and may alias with globally visible data. Such stores are identified through detailed pointer analysis. We provide three solutions to handle statically unknown memory aliasing: (1) a hardware solution employing Hardware Transactional Memory (HTM) in an innovative manner for sequential applications; (2) a purely software approach, to ensure generality and compatibility with architectures that do not provide support for HTM; and, finally, (3) a conservative approach where the original code is invoked instead.

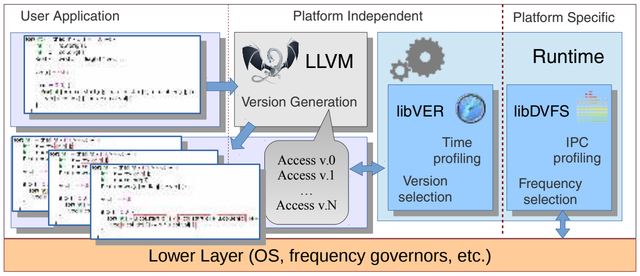


Figure 1: The overall system design demonstrating all system components to support multi-versioning

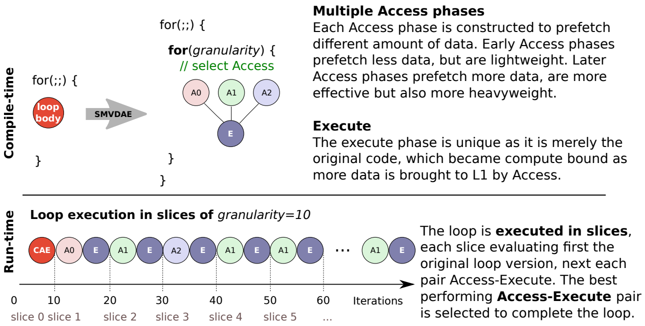
Typically, DVFS techniques aim to reduce energy consumption, while minimizing the negative impact on performance or maintaining performance at best. In contrast, we show that complex, general-purpose applications not only exhibit *energy improvements, but also significant performance boosts*, when executed under the decoupled access-execute model. Despite the partial code duplication, decoupled execution exposes more memory-level parallelism (by hoisting all loads to the access phase), hides latency (by separating the load operation –access– from the use –execute–) and increases instruction level parallelism (by invoking the execute version when the required data is available in the cache). While the behavior of complex applications is challenging for static analysis and for hardware predictors, software multi-versioning DAE (SMV-DAE) improves energy delay product (EDP), by over 20% on average (over 70% peak), and provides energy benefits of over 30% for memory-bound general-purpose applications. Furthermore, SMV-DAE is a **self-adapting** technique, automatically selecting the best performing access version at runtime, and a **self-healing** technique, resorting to the original code version for applications which are not amenable to decoupling (compute-bound applications), hence the *race-to-sleep* technique [44] is applied instead.

2. Overall System Design

Our SMVDAE framework, depicted in Figure 1, consists of a compiler infrastructure, which generates multiple access versions, and two libraries to measure the impact of each version to performance (libVER) and energy (libDVFS) respectively. LibVER profiles the performance of each access-execute pair and selects the most efficient Access phase, while libDVFS identifies and applies optimal [1] DVFS settings for each phase of the previously selected pair (by libVER).

The mechanism to generate multiple access versions is implemented as an LLVM [28] compiler pass. Functions on the critical path are marked as a target for SMVDAE execution, identified either via offline profiling or based on programmer's annotations. The compiler then automatically prepares the outermost loops contained in these functions to be run in *slices*, as shown in Figure 2. A slice represents a subset of consecutive iterations of the outermost loop and its size is referred to as *granularity*. Once the loop is split in slices, the granularity is adjusted experimentally such that the workload of each slice fits in the private cache of the core. Next, the access phases $\{A_0, A_1, .., A_n\}$ and the execute phase E are generated per slice. The execution model is illustrated in Figure 2: the first loop slice is executed in the original version (CAE), as a

[1] The selection of *optimal* frequency can be performed with respect to either performance, energy, or EDP. In our work we opt for EDP and we use similar techniques as presented by Spiliopoulos et al. [38]



Multiple Access phases
Each Access phase is constructed to prefetch different amount of data. Early Access phases prefetch less data, but are lightweight. Later Access phases prefetch more data, are more effective but also more heavyweight.

Execute
The execute phase is unique as it is merely the original code, which became compute bound as more data is brought to L1 by Access.

The loop is **executed in slices**, each slice evaluating first the original loop version, next each pair Access-Execute. The best performing **Access-Execute** pair is selected to complete the loop.

Figure 2: SMVDAE compilation and execution model: The compiler generates multiple Access versions which are evaluated at runtime and the best performing Access-Execute pair is selected to complete the loop execution.

```
for(int i = thedim - 1; i >= 0; i--) {
    int    r = row.orig[i];
    int    c = col.orig[i];
    Real x = wrk[c] = diag[r] * vec[r];

    vec[r] = 0.0;

    if (x != 0.0) {
      for(int j = u.col.start[c]; j < u.col.start[c] + u.col.len[c]; j++)
        vec[u.col.idx[j]] -= x * u.col.val[j];
    }
  }
```

Figure 3: Code snippet of statically challenging code extracted from `soplex`, SPEC CPU 2006. The compiler cannot clearly determine which memory accesses to prefetch to achieve the right balance between *overhead* (for computing the address) and *benefits* (by prefetching the data to the L1 cache).

reference for performance and energy; the access phase A_0 of the next loop slice is run, followed by the execute phase of the same slice; next, the DAE run of the following slice is invoked with the pair A_{i+1} and E, until all access phases have been evaluated. The access phase that performs best is selected for the DAE run of the following slices until the entire loop completes its execution. Our approach requires loops with a considerable number of iterations to enable the evaluation of a sufficient number of Access versions and to compensate for the selection overhead. For loops with low loop-trip counts, the programmer may explicitly select the best version (e.g., after profiling). Orchestrating the execution of Access-Execute pairs per slice can be handled with state of the art runtime systems, such as Protean code [27], VMAD [20, 22], etc.[2].

Evaluating the access versions on successive slices introduces negligible overhead, as the loop progresses naturally. Optimizing the version selection at runtime is not the main focus of this work. Therefore, the best performing version is selected once in the beginning of the loop. This simple technique provides good results (on the evaluated benchmarks) and minimizes the selection overhead. Furthermore, loop slicing is performed at coarse granularity which covers the potential heterogeneity of the loop behavior. Note that, in most cases the slicing is performed on the outermost loop, however, in case the workload of a single loop iteration overflows the private cache, the child loop is sliced instead.

The execution time of each Access and Execute phase is measured by libVER library. The execution-time profiling requires only a low-latency timer, available on every contemporary system, hence we refer to this library as being system independent. Energy measurements are performed using libDVFS library, following a similar methodology as in our previous work on decoupled access-execution [21, 26]. LibDVFS uses a linear Instructions-per-Cycle (IPC) power model (specific to each architecture) to estimate the energy for each Access and Execute phase separately. This infrastructure allows us to measure efficiency as $Energy\ Delay\ Product\ (EDP) = Time_{total} \times Energy_{Total}$ [12, 24] which accounts both for performance and energy.

3. Methodology

SMVDAE targets loops with complex control flow, memory accesses performed via indirections and pointer chasing, typically applications which pose challenges for compile-time optimizations.

We proceed with a pedagogical example, shown in Figure 3, for which the compiler cannot statically estimate the overhead of computing the addresses with many indirections (e.g. u.col.start[c], vec[u.col.idx[j]]) or guarded by conditionals, and the benefit of prefetching such addresses. Figure 4 illustrates these trade-offs when various memory addresses are selected for prefetching. Selection is based on the level of indirection, lower levels of indirection generate lightweight, but less efficient access phases, while higher levels of indirection have the potential to prefetch addresses outside the reach of the hardware prefetcher, but at a higher cost. It is our goal to provide compile-time support to generate the most promising access versions and select at runtime the one that provides the best balance between overhead and benefits.

Section 3.1 provides the general approach for generating an access phase starting from a list of target loads. Next, Section 3.2 details how the target loads are selected, which lead to multiple, customized access versions (one version for each set of loads).

3.1 Access Phase Generation

As the role of the access phase is to *prefetch* data required by the execute phase, the access version is a simplified clone of the original code, containing only address computation and instructions maintaining the control flow. A key aspect is that variables written within the access phase do not escape its scope, meaning that no writes to globally visible data are allowed (*i.e.* global variables and function arguments [3], henceforth referred to as “globals” for brevity), since each loop slice is re-executed by the execute version, which performs the writes. Consequently, the compiler must ensure that globals are correctly identified and no updates are performed within the access phase. In contrast, updates of local variables are not only allowed, but also mandatory for preserving the control-flow. This requires a precise pointer analysis. The algorithms for selecting instructions for the access phase control flow (3.1.1), and memory address computations (3.1.2) are described below. All other instructions are removed from the clone, generating a lean access version.

3.1.1 CFG Skeleton

The CFG skeleton represents the minimal set of instructions that duplicates the control flow of the original slice. Algorithm 1 describes how these instructions are collected. The algorithm first adds all terminator instructions —conditional / unconditional branches, exit, return instructions— to set K (step 1), and then processes each instruction from the set by adding all its requirements (steps 2 – 3). Requirements are found recursively using the

[2] The runtime version selection mechanism is orthogonal to this proposal and it can be replaced by more advanced heuristics, which, for instance, might re-evaluate promising access phases to better serve different execution phases.

[3] Conservatively, all function arguments are handled as potentially escaping the scope of the access phase.

Prefetching 1^{st} level of indirections: Lean address computation, but possibly already prefetched by hardware.

```
for(int i = thedim - 1; i >= 0; i--) {
   int   r = row.orig[i];
   int   c = col.orig[i];
   Real x = wrk[c] = diag[r] * vec[r];

   vec[r] = 0.0;

   if (x != 0.0) {
     for(int j = u.col.start[c]; j < u.col.start[c] + u.col.len[c]; j++)
       vec[u.col.idx[j]] -= x * u.col.val[j];
   }
 }
```

Prefetching 2^{nd} level of indirections: prefetches more of the useful data. Note that the addresses in the inner loop, e.g. `u.col.start[c]` are computed with three indirections (not by computing `u.col.start`).

```
for(int i = thedim - 1; i >= 0; i--) {
   int   r = row.orig[i];
   int   c = col.orig[i];
   Real x = wrk[c] = diag[r] * vec[r];

   vec[r] = 0.0;

   if (x != 0.0) {
     for(int j = u.col.start[c]; j < u.col.start[c] + u.col.len[c]; j++)
       vec[u.col.idx[j]] -= x * u.col.val[j];
   }
 }
```

Prefetching 3^{rd} level of indirections: Increased address computation overhead, may or may not lead to runtime benefits.

```
for(int i = thedim - 1; i >= 0; i--) {
   int   r = row.orig[i];
   int   c = col.orig[i];
   Real x = wrk[c] = diag[r] * vec[r];

   vec[r] = 0.0;

   if (x != 0.0) {
     for(int j = u.col.start[c]; j < u.col.start[c] + u.col.len[c]; j++)
       vec[u.col.idx[j]] -= x * u.col.val[j];
   }
 }
```

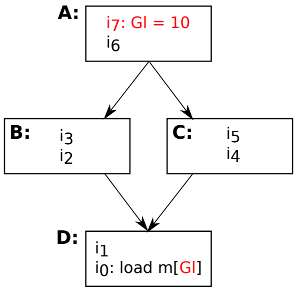
Prefetching 4^{th} level of indirections: Heavier access phase, prefetches data difficult to reach by the hardware prefetcher. However the cost of maintaining the control flow and computing the addresses may or may not be hidden by the benefits.

```
for(int i = thedim - 1; i >= 0; i--) {
   int   r = row.orig[i];
   int   c = col.orig[i];
   Real x = wrk[c] = diag[r] * vec[r];

   vec[r] = 0.0;

   if (x != 0.0) {
     for(int j = u.col.start[c]; j < u.col.start[c] + u.col.len[c]; j++)
       vec[u.col.idx[j]] -= x * u.col.val[j];
   }
 }
```

Figure 4: It is impossible for the compiler to optimally balance address computation overheads and benefits for the general case.



Starting from the load i_0: `load m[Gl]`, we parse the instructions on the reversed control flow graph and identify stores that alias the address of the load. First, instructions in the current basic block are parsed (i_1 from **D**), followed by instructions in the predecessors basic blocks (i_2, i_3 from **B** and i_4, i_5 from **C**), and the algorithm continues processing predecessor basic blocks (**A**), until an aliasing store is found (i_7: `Gl = 10`) or all preceding instructions have been processed.

Figure 5: Algorithm for identifying stores aliasing the loads selected for inclusion in the access phase.

use-def chain and the algorithm stops when no new instructions are added to the set (step 4).

Algorithm 1 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. (Algorithm 2 describes how to find the
   store instructions.)
4. Repeat steps 2 and 3 until no choice of $I$ will cause further
   instructions to be added to $K$.
```

Store instructions are selected by "climbing up" the reversed control flow graph and checking whether the store and load addresses alias. Aliasing is determined using the LLVM *AliasAnalysis* pass. Specifically, alias matches of the types *MustAlias*, where the addresses reference the same memory area, and *PartialAlias*, where the referenced memory overlap, are considered [4]. Algorithm 2 details how the search is conducted and Figure 5 provides an example.

Once the instructions building up the CFG are identified, the compiler must ensure the access phase is side-effect free or discard it otherwise. The access version is discarded if the CFG depends on (i) function calls (not annotated as read-only) or (ii) there exist at least one store instruction aliasing with a non-local value. To identify local values, the pass analyzes memory allocation. Local pointers in LLVM IR are allocated by a special allocation instruction (*AllocaInst*). The pointers can pass through a number of instructions before being used by a memory operation, most notably an indexing operation (*GetElementPtrInst*) and various types of casts (*CastInst*). If the input pointer to any of these types of instructions is local, the output pointer is considered local as well. Conservatively, all pointers stemming from other sources (e.g. global variables, function arguments) are considered global, or external to the function. Store instructions targeting the latter pointers are prohibited in the access phase.

3.1.2 Memory Address Computation

The purpose of the access phase is to prefetch data to the cache to be ready for use once the execute phase takes over. As phases do not share local data, it is only useful to prefetch global variables. While it is possible to prefetch all global variables, doing so may lead to

[4] *MayAlias* matches are not included, to keep the Access phase lightweight and efficient. The decision was guided by the observation that in practice *MayAlias* materializes to *NoAlias*.

Algorithm 2 Set S collects potential sources (store instructions) for a load instruction, L, loading from address A.

```
1. Starting from $L$, search backwards through the instruction list
   of the parent function.
2. For each instruction $I$, check
   (a) If $I$ has been encountered before, cancel the rest of step 2
       and go to step 4.
   (b) If $I$ is a store instruction to address $B$, check
       i.  If $A$ and $B$ alias exactly (MustAlias), add $I$ to $S$, cancel
           the rest of step 2 and go to step 4.
       ii. Otherwise, if $A$ and $B$ alias partially (PartialAlias), add
           $I$ to $S$ and continue.
   (c) If $I$ is the first instruction in the basic block, go to step 3.
   (d) Otherwise, continue step 2.
3. Enqueue preceding basic blocks in the CFG for subsequent
   search.
4. If there are no queued basic blocks, stop. Otherwise, pick the
   first queued basic block and go to step 2, searching the basic
   block backwards from its last instruction.
```

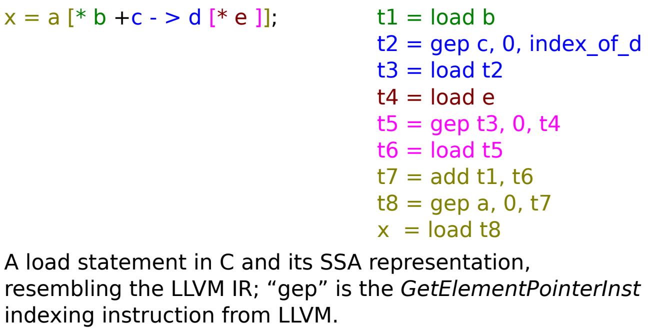
very heavy access phases. Instead, only a subset of all loads[5] are prefetched, as detailed in Section 3.2.1. Loads are replaced with prefetch instructions, unless the loaded value is further required in the access phase. Moreover, the prefetch instruction is inserted right after the address computation, not necessarily in the original position of the load. Thus, only one prefetch instruction is inserted even if several loads target the same address. An algorithm similar to the one building the CFG is used. First, the set of target loads is built (Section 3.2.1), equivalent to the starting set K in Algorithm 1 and next, all required instructions are added following the same algorithms as in Section 3.1.1. Unlike the algorithm for building the CFG, access phases are not discarded in case a memory address computation relies on a store to a global value. Instead, only the corresponding prefetch instruction is discarded, together with its required instructions.

3.1.3 Removal of Superfluous Code

Once the instructions for building the access phase are selected, we run the classical compiler optimizations on the generated access version. In particular, dead code elimination and optimizations aimed to simplify the control flow ensure that empty basic blocks and unnecessary branches are removed. The entire CFG skeleton is first inserted in the basic access version to ensure that all loads are reachable and that the loop's control flow is preserved. However, part of the CFG skeleton might not be necessary, if the basic blocks do not contain instructions required for computing a target address or for reaching a prefetch instruction. Also, prefetch instructions which accompany a load required in the access phase are removed in this step.

3.2 Multi-Versioning

To adjust the balance between cost and efficiency, multiple access versions are created, each employing its own policy for selecting the target loads for prefetching. The heuristics employed in this proposal to decide the set of target loads are based on the number of indirections required to calculate the address to be prefetched. More precisely, a threshold T_i is set for each access version, and only addresses that require less that T_i indirections are prefetched in the associated access version A_i. For instance, address `a[b[i]]` requires one indirection, hence we will generate A_0 prefetching addresses that require $T_0 = 0$ indirections, `b[i]`, and A_1 with $T_1 = 1$ prefetching `a[b[i]]`.



```
x = a [* b +c - > d [* e ]];          t1 = load b
                                      t2 = gep c, 0, index_of_d
                                      t3 = load t2
                                      t4 = load e
                                      t5 = gep t3, 0, t4
                                      t6 = load t5
                                      t7 = add t1, t6
                                      t8 = gep a, 0, t7
                                      x  = load t8
```

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

Figure 6: The number of indirections of an address A is given by the number of loads required to compute A.

3.2.1 Indirection Measure

We define "indirection" as a metric indicating how many intermediate loads are required to compute an address. Figure 6 shows an example which converts a C statement that loads a value to the equivalent static single assignment (SSA) representation, resembling the LLVM intermediate representation. In this example, the indirection measure for the *load* t_8 to `x` is four: $t_1,\ t_3,\ t_4,\ t_6$.

The general algorithm to find the indirection measure for an arbitrary address A follows the same steps as Algorithm 1 to identify the set of instructions for computing address A.

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 1, using $D$ in place of $K$.
3. The level of indirection is the number of load instructions
   present in $D$.
```

Note that two access versions A_i and A_j can be identical, despite having distinct thresholds for the maximal number of allowed indirections, $T_i < T_j$, if there is no address that requires n loads, $T_i < n \leq T_j$. For instance, $Access_2$ in Figure 6 contains instructions $t_1 - t_6$, as t_6 requires loads t_3 and t_4, hence a maximum indirection of two. $Access_3$ contains precisely the same instructions, as there is no memory access that requires three indirections, yielding $Access_2$ and $Access_3$ identical. Finally, $Access_4$ differs, as it can reach load t_8, which has an indirection of four.

3.2.2 Generating Multiple Access Versions

SMVDAE generates one access version for each level of indirections. For the code example in Figure 4, SMVDAE generates the access versions presented in Figure 7. $Access_0$ displays also the instructions before removing the superfluous code. These prefetch instructions are removed because the corresponding loads are required and kept in the access phase. Note that although $Access_2$ and $Access_3$ prefetch only addresses in the innermost group, a significant part of the control flow is replicated to reach the prefetch instruction. It is therefore difficult to estimate statically the actual benefit of prefetching these addresses.

[5] DAE prefetches loads only, since stores are not on the critical path.

Access 0:

```
for(int i = thedim - 1; i >= 0; i--) {
   ~~prefetch orig[i];~~
   int r = row.orig[i];
   int c = col.orig[i];
   ~~prefetch wrk[r];~~
   ~~prefetch diag[r];~~
   ~~prefetch vec[r];~~
   Real x = wrk[c] = diag[r] * vec[r];
   if (x != 0.0) {
     ~~prefetch u.col~~;
     ~~prefetch start[c]~~;
     ~~prefetch len[c]~~;
     for(int j = u.col.start[c]; j < u.col.start[c] + u.col.len[c]; j++){
       prefetch idx[j];
       prefetch val[j];
     }
   }
 }
```

Access 1:

```
for(int i = thedim - 1; i >= 0; i--) {
   prefetch row.orig[i];
   prefetch col.diag[r];
}
```

Access 2:

```
for(int i = thedim - 1; i >= 0; i--) {
   Real x = wrk[c] = diag[r] * vec[r];
   if (x != 0.0) {
     for(int j = u.col.start[c]; j < u.col.start[c] + u.col.len[c]; j++){
       prefetch u.col.idx[j];
       prefetch u.col.val[j];
     }
   }
 }
```

Access 3:

```
for(int i = thedim - 1; i >= 0; i--) {
   Real x = wrk[c] = diag[r] * vec[r];
   if (x != 0.0) {
     for(int j = u.col.start[c]; j < u.col.start[c] + u.col.len[c]; j++)
       prefetch vec[u.col.val[j]];
   }
 }
```

Figure 7: SMVDAE generated access versions, one version for each level of indirections.

3.3 Statically Unknown Memory Dependences

As described in Algorithm 2, the access phase includes stores that *MustAlias* or *PartiallyAlias* with other loads required for the control flow or address computation, as long as such stores do not alias with global variables [6]. In case they do alias, the Access phase is discarded, i.e. the code is not decoupled, since it cannot be guaranteed to be side-effect free. To enable DAE when stores in the access phase may alias global variables, techniques are required to ensure that all global variables are returned to the state they had before the invocation of the access phase. Although our experiments show that this problem rarely occurs in practice (1 out of 14 evaluated benchmarks), we provide a generic solution to decouple general purpose applications. We perform a pointer analysis of the generated access phase and mark as "unsafe" (using metadata information) stores that alias global variables. We provide two solutions to ensure that unsafe stores are harmless (explained in Sections 3.3.1 and 3.3.2).

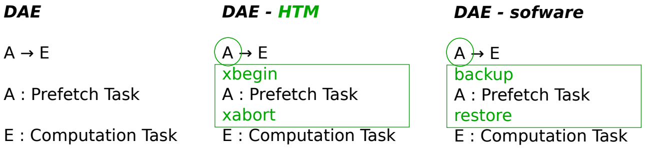


Figure 8: If a store in Access aliases a global variable, we provide two solutions: (1) Embed Access in a hardware transaction to ensure side-effect free access phases; (2) Guard Access with a *backup-and-restore* mechanism to preserve the original value of variables that escape Access scope.

3.3.1 HTM

On architectures that provide support for Hardware Transactional Memory (HTM), e.g. Intel Haswell [13], we embed the access phase in a transaction using Restricted Transactional Memory (RTM) [2, 17]. This is done by inserting a forced XABORT at the end of the access phase, and specify the execute version as the fallback path. Thus, XABORT ensures that values loaded or prefetched during the access phase are available in the cache, while the written cache lines are invalidated, yielding the unsafe stores harmless. We adjust the *granularity* of the slice to fit the transaction buffer and avoid unexpected capacity aborts. Nevertheless, when a transaction aborts before the access phase completes its execution, we simply continue with the execute phase, since part of the data is already prefetched in the L1 cache.

3.3.2 Software Managed Stores

A software solution to this problem is to safeguard the global variables at compile-time by copying them to local variables and restoring their original values before exiting the access phase. Should the target address of the store depend on the loop index, the overhead of allocating memory and saving a back-up copy in *each iteration* would be too large and would hide any potential benefit. Therefore, we implement a light version of the backup-and-restore mechanism, only applicable when the address to be protected is loop-independent and therefore can be hoisted outside the loop. Otherwise, the access phase is discarded.

4. Experimental Setup

We perform our evaluation on two different generations of Intel desktop processors. A *Sandybridge* architecture (Intel® Core^{TM} i7-2600K CPU, with a frequency range from 1.6GHz to 3.40GHz), and a *Haswell* architecture (Intel® Core^{TM} i7-4790K CPU, with a frequency range from 800MHz to 4.00GHz), that features Hardware Transactional Memory (HTM). Both machines are equipped with 16 GB DDR3 RAM running at 1333MHz, and have a similar cache hierarchy: 32 KB private (per core) L1 instruction and data caches, 256 KB private (per core) L2, and a shared 8192 KB L3. Haswell, in addition, has various improvements in the microarchitecture and in the hardware prefetcher.

Energy is estimated separately for each phase, with the use of a linear IPC power model. The model accuracy is verified against measurements performed on real hardware (with an average error of $\approx 3\%$) using a methodology similar to previous work [21, 25, 26, 38]. The instruction count at each phase is performed with the help of PAPI [32] library, while cycles are counted with the use of the *rdtsc* x86 instruction. The Instructions-per-Cycle (IPC) model enables us to estimate the energy expenditure of each phase at each frequency and therefore, to predict the optimal frequency. Typically, a low or intermediate frequency is selected for the access phase, and a higher frequency for the execute phase, depending on the memory boundness of each phase (as determined by its IPC).

[6] By "global variables" we refer to variables whose scope escapes the Access phase

Table 1: Selection of benchmarks (BM) from SPEC CPU 2006 (up) and Parboil (bottom). The functions on the critical path were selected to evaluate SMVDAE.

BM	File	Function
astar	Way2_.cpp Way_.cpp	way2obj::releasebound wayobj::makebound2
bzip	compress.c decompress.c	generateMTFValues BZ2_decompress
h264ref	mv-search.c mv-search.c	SetupFastFullPelSearch BlockMotionSearch
hmmer	fast_algorithms.c	P7Viterbi
lbm	lbm.c	LBM_performStreamCollide
libQ	gates.c gates.c gates.c	quantum_toffoli quantum_sigma_x quantum_cnot
mcf	pbeampp.c	primal_bea_mpp
milc	m_mat_na.c	mult_su3_na
soplex	ssvector.cc spxsteeppr.cc ssvector.cc	SSVector::assign2product4... SPxSteepPR::entered4X SSVector::setup
sphinx3	cont_mgau.c	mgau_eval
cutcp	cutcpu.c	cpu_compute_cutoff_potent...
mri-g.	CPU_kernels.c	gridding_Gold
sad	sad_cpu.c	sad4_cpu
stencil	kernels.c	cpu_stencil

Table 2: *Gran.* is slice granularity (in loop iterations), *#Versions* corresponds to the total number of distinct access phases per application, *M.I.* is the max depth of indirection in an access phase, *B.I.* is the level of indirections in the best performing access phase, and *#Slices* is the number of slices executed for each benchmark. *B.I.* values correspond to the execution shown in Figure 12 on an Intel i7-4790K CPU. For *bzip* and *hmmer* the original (CAE) version was selected.

BM	Gran.	#Versions	M.I.	B.I.	#Slices
astar	300	6	6	1	21028729
bzip	2048	9	10	CAE	382475892
h264ref	512	12	12	0	42986240
hmmer	75	7	6	CAE	4944811
lbm	192	2	2	2	26001000
libQ	1450	3	3	3	94107092
mcf	4	3	3	3	21854886
milc	1650	3	2	2	460800000
soplex	300	13	15	12	22823811
sphinx3	256	7	7	7	599989032
cutcp	2048	7	19	11	56
mri-g.	16	5	10	9	165995
sad	16	3	6	0	5
stencil	4	2	2	0	2048

We evaluated SMVDAE on a selection of applications from SPEC CPU2006 [15] and from Parboil [39] benchmark suites, evaluating both memory- and compute-bound applications. The classification was guided by two metrics, cache miss rate and Cycles per Instruction (CPI), as reported in previous work [18, 33]. While memory bound applications are a good target for DVFS techniques, and therefore for SMVDAE, compute-bound applications do not benefit from adjusting the frequency, nevertheless, we emphasize that SMVDAE does not harm their performance. Among the memory bound applications we selected *mcf*, *milc*, *soplex* and *lbm*, expected to benefit from SMVDAE, *libquantum*, *astar*, *sphinx3* and *sad* are medium compute- and memory-bound, while *bzip*, *hmmer*, *cutcp*, *mri-g.* and *stencil* are compute bound, hence not a target for DVFS techniques. For each benchmark, we applied SMVDAE on the functions on the critical path, which for SPEC CPU-2006 were identified based on previous studies [1], while for Parboil we performed an initial profiling to determine the critical functions. The target functions are listed in Table 1.

We have conducted our experiments using the largest input set, *ref*, for SPEC CPU 2006 and evaluated all available *ref* inputs when several were provided (e.g. for *soplex*, *hmmer* and *astar*). Parboil benchmarks were evaluated with *large* datasets. Thus, the workloads evaluated in this proposal contain loops with high loop trip counts, which enabled the generation of coarse-grain Access phases and hid selection overheads. Finally, *granularity* was set experimentally. Other proposals [40] estimate the workload of a loop iteration and can be used to set the *granularity* automatically at compile-time, such that the workload of each slice fits the private L1d cache. This study is orthogonal to our approach.

Table 2 indicates the number of generated access versions per application, which is less or equal to the maximum number of indirections found in the benchmark. Each version $Access_n$ is generated by setting the threshold $T_n = n$ for the number of indirections, however if two consecutive access versions are identical (see Section 3), only one is included. The results clearly emphasize that some applications benefit from prefetching deeper indirections (*soplex, mri-g*), others show better results when the lower levels of indirection are used (e.g. *bzip, astar, hmmer, stencil*, while some applications obtain benefits with a medium indirection depth (*cutcp*).

5. Evaluation

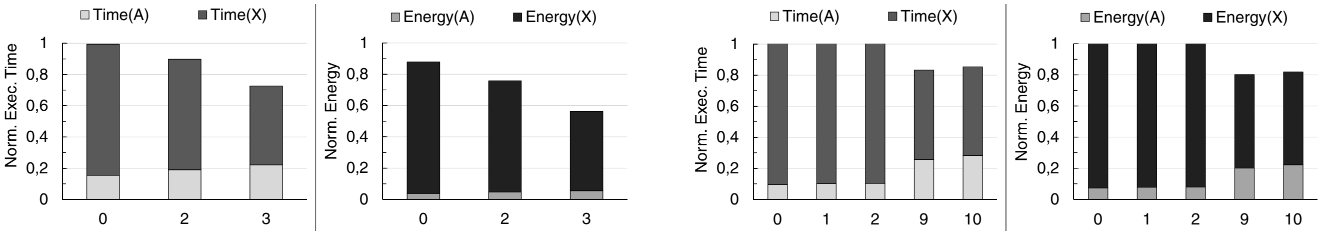
5.1 Multi-Version Case-Study

This section focuses on two case study applications, *mcf* and *mri-g*, and presents the performance and energy benefits of all versions, as shown in Figure 9. For mcf, we observe that performance increases as we increase the number of indirections in the access phase and energy follows the same trend. $Access_3$, which prefetches the deepest level of indirections, achieves the best performance. An important observation is that even when the overall performance is unaffected e.g., for *mcf* $Access_0$, there is still potential for energy savings, if a significant amount of the total execution time is within the access phase.

For *mri-g*, we observe that the first three versions bring no performance or energy improvement, while the last two versions yield 17–18% performance and $\approx 20\%$ energy improvements. Although the access phases of the two applications (*mcf* and *mri-g*) represent a similar percentage of the total execution time, the energy savings registered for *mri-g* are significantly lower compared to *mcf*. This is a consequence of selecting different optimal frequencies for the access phases for each application, namely 1.7GHz for *mcf*, and 3.4GHz for *mri-g*.

5.2 Dynamic Application Behavior

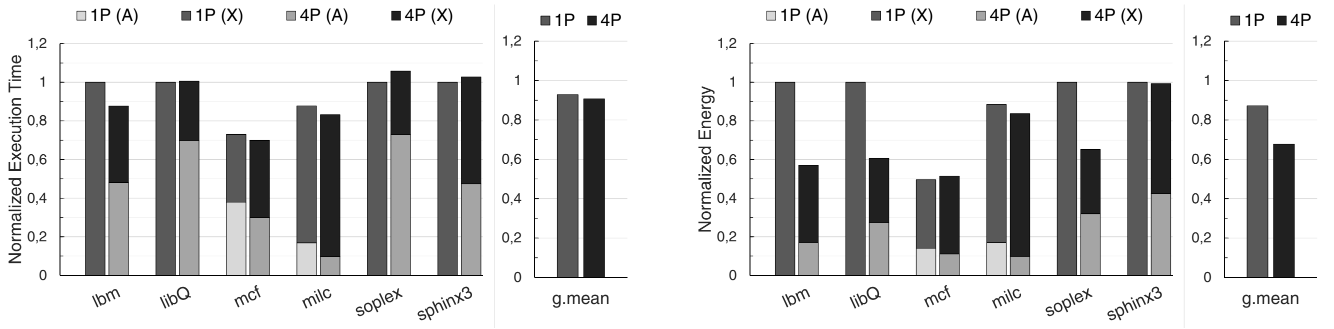
The main goal of multi-versioning is to adapt and therefore, run optimally on different architectures and under different system loads. This section evaluates a memory-bound subset of SPEC-CPU2006 on the same architecture (Intel i7-2600K), under different system loads —for example, when a single instance of an application runs in isolation in the system, and when multiple instances (e.g., 4) of the same application compete for shared resources (e.g., last level cache and memory bandwidth). As expected, SMVDAE provides



(a) Mutli-version performance/energy for mcf

(b) Mutli-version performance/energy for mri-g

Figure 9: Normalized execution time and energy for all generated versions for *mcf*, and *mri-g*. Ox axis displays the maximum number of indirections per version. Experiments were conducted on an Intel i7-4790K CPU.



(a) Execution time for different system loads

(b) Energy for different system loads

Figure 10: Normalized execution time and energy for a selection of SPEC 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).

higher performance/energy improvements when the memory system is under heavy strain (multi-instance runs), as shown in Figure 10. For the selected (memory-bound) benchmarks, we observe that when they do not run in isolation *a DAE version always outperforms* the original (CAE version), yielding on average 10% performance and over 30% energy improvements.

The variance in performance and energy observed for the same application under different system loads, further motivates the use of software multi-versioning. Even if one could statically infer the optimal version for a specific architecture, e.g., using offline profiling, the actual optimal version might differ at runtime based on the system load, therefore requiring a dynamic approach. In Figure 10 we observe that *mcf* and *milc* benefit from SMVDAE even when they run in isolation (although different access versions are selected when running one process (1P) versus four processes 4P). This is not the case for the rest: *lbm*, *libQ*, *soplex*, and *sphinx3*, where SMVDAE yields good energy improvements in 4P runs, while in 1P runs the original (CAE version) is selected.

5.3 Performance and Energy Evaluation

We evaluate SMVDAE on two different systems (Intel i7-2600K and Intel i7-4790K) using three metrics: performance, energy and EDP. The evaluation is performed with 4 instances of the same application pinned on different cores stressing the memory bandwidth. Our first observation is that DAE yields higher overall improvements on the i7-4790K over the i7-2600K, despite the fact that both processors use similar memory modules (same frequency and timings). The reason is the performance gap between CPU and memory (i7-4790K runs at 4GHz while i7-2600K at 3.4GHz). On average we observe 20–25% EDP improvements and 16–22% energy improvements (over 5% performance). For predominantly compute-bound applications like *bzip* and *hmmer* the original version was selected in both architectures. For the rest of the compute-bound applications, we observe small improvements from SMVDAE, while for the memory-bound ones (*lbm*, *libQ*, *mcf*, *milc* and *soplex*) we observe significant EDP and energy savings. We measure over 40% EDP and energy savings on average (for memory bound applications), with a peak of 72% EDP improvement for *mcf*. Finally, for *sphinx3* we observe that despite spending a significant amount of the total execution time within the access phase, the energy/EDP savings are minimal, due to the heavy calculations required in the access phase that suggest a high optimal frequency.

5.4 SMVDAE Code-Size Overhead

Code explosion is a typical concern associated with multi-versioning techniques, however SMVDAE does not incur such problems since it only targets functions on the critical path and imposes a threshold on the number of generated versions, if necessary. Prefetching very deep indirections (i.e. with an indirection level higher than , for example, 20) is unlikely to bring benefits, due to the overhead of computing the address. In our experiments we have generated and evaluated all possible access versions, namely until the deepest indirection level. Since in many cases consecutive access phases are equivalent, as explained in Section 3 and exemplified in Figure 9, the final number of generated access versions (e.g., soplex has 13 versions) may be lower than the deepest indirection level (soplex has a maximum indirection depth of 15). Since the critical functions, although dominating the execution time, represent a small fraction of the total code size, the overhead of SMVDAE with respect to code size is negligible, less than 1% for most of the applications, except h264ref (2.6%, including 12 versions) and soplex (1.3%, with 13 versions).

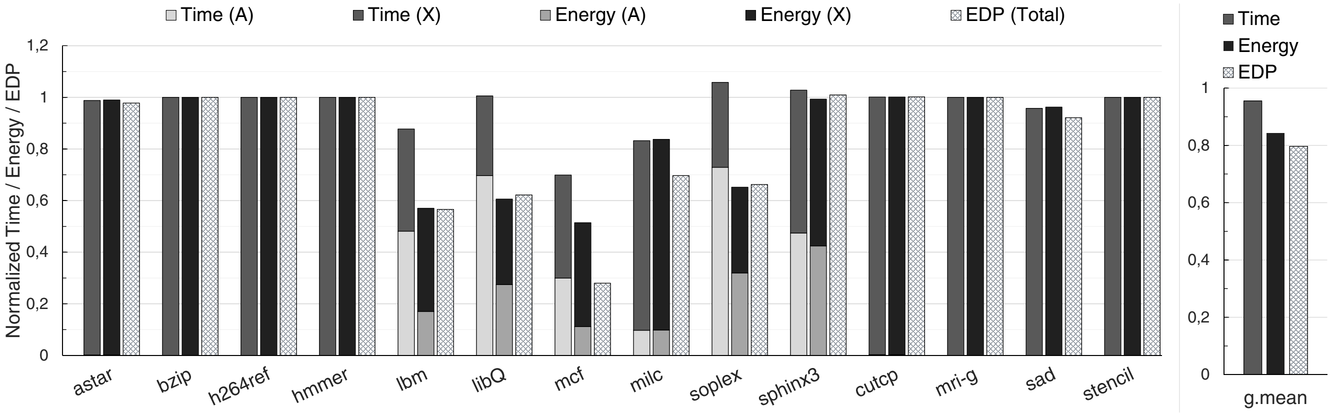


Figure 11: Overall improvements in performance, energy, and EDP on Sandybridge Intel i7-2600K

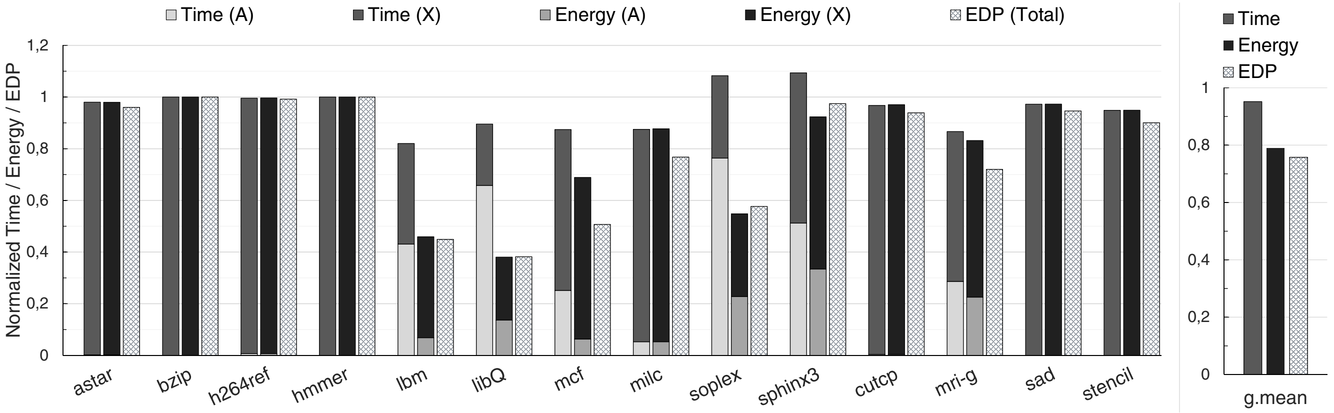


Figure 12: Overall improvements in performance, energy, and EDP on Haswell Intel i7-4790K

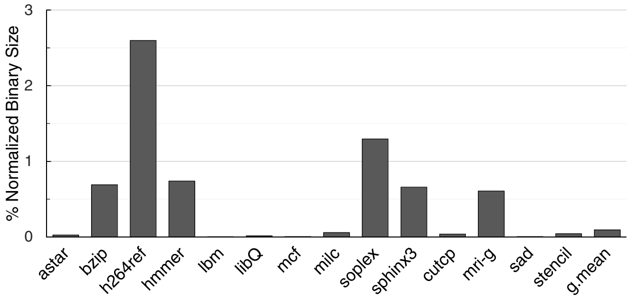


Figure 13: % Code-size overhead due to multi-versioning (up to 20 versions).

5.5 Software and Hardware Solutions to Memory Aliasing

In practice, the compiler was successful in statically disambiguating the memory accesses and in generating side-effect free access versions. Among the evaluated benchmarks, only a subset of the *bzip* access phases contained stores that may-alias globally visible variables. However, including such stores in the access phases generated prohibitively heavy versions, for this particular application, hence the original CAE version was automatically selected to complete the execution. Embedding the "unsafe" access phases in a hardware transaction or using the backup-and-restore mechanism in software adds an additional overhead. HTM for instance may introduce overall performance penalties ranging from 0% to 60% in total, depending on the number and granularity of *slices* protected by a transaction. On the other hand, HTM might also abort ahead of time, in which case there is an apparent speed-up compared to the decoupled version, due to an incomplete access phase. We did not investigate these techniques in depth, as the need did not occur in practice. Nevertheless, "unsafe" access versions are transformed to safe versions and made available for the runtime version selection, should they lead to a more energy efficient execution.

6. Related Work

State of the art compile-time optimizations targeting energy efficiency rely on statically injecting instructions for scaling voltage or frequency [42], [6] at runtime. This proposal builds upon the decoupled access-execute scheme [21], which is a pioneering work in performing aggressive compile-time code transformations that remodel the application behavior to better adapt to the underlying hardware.

The decoupled access - execute model was initially proposed in hardware by Smith [37]. The computation units were only aware of the operation and the operands were given through queues. Koukos et al. [26] proposed a decoupled access-execute (DAE) technique in software, using the private caches of the core as communication channels between Access and Execute. DAE exploits a potential for energy efficiency optimizations in task based parallel applications, while further work by Jimborean et al. [21] extends these ideas and provides a concrete compiler methodology to support the automatic decoupled access execute model. This work [21] relied on polyhedral transformations and aggressive code simplifications to generate low-overhead (lightweight) access phases for scientific codes. However, these approaches are not typically applicable to general purpose serial applications, whose pointer-chasing, dynamic data structures and conditionals hinder static analysis, or result in either inefficient or high-overhead access phases. SMVDAE addresses this problem by providing efficient means to adjust the trade-off and the benefits of prefetching, at runtime.

Generating skeleton phases derived from the original code resembles techniques such as inspector-executor [3, 5, 35, 36, 43] and helper thread [23, 34, 45] methods. Inspector-executor techniques [3, 5, 35, 36, 43] rely on the concept of running an inspector version aimed to instrument and analyze the code and typically includes extra code to process the collected information. Next, the executor is optimized based on the outcome of the inspector code. This technique is commonly used to overcome static limitations, e.g. analyze dependence patterns that could not be explored statically by a compiler and optimize or parallelize the executor accordingly. Helper threads are designed to accelerate the performance of single threaded applications [23, 34, 45] using a prefetching thread that runs ahead and warms up the cache for the worker thread. Such methods require that the inspector code or the prefetching thread, respectively, are generated statically, which exhibits similar limitations to the previously proposed DAE. Should memory accesses and computation be entailed, the generated code would incur a large overhead, cancelling the benefits of the method.

Software multi-versioning was proposed to reduce the overhead of code instrumentation [4, 8, 10, 14, 16, 30], for checking program correctness [14], for loop parallelization, for automatic, speculative optimizations [7, 19, 29], or to optimize the execution for different inputs [41, 46]. The technique periodically switches between high-overhead instrumenting versions and more efficient original or optimized versions [4, 8, 10, 16]. Instrumentation conclusions are then used to guide optimizations [7, 29], speculative parallelization [19] or to detect data races [30]. SMVDAE relies on multi-versioning not for diminishing the overhead or for predicting application behavior, but for finding the right balance between cost and benefit, using runtime information.

7. Conclusions

We propose a compiler technique to improve the energy efficiency of complex, general-purpose applications, which are typically not amenable to static analysis. Our solution gracefully blends DVFS [11, 25, 31, 38], employing the decoupled access-execute model for memory-bound applications, and race-to-sleep [44] methods, running the original version under maximum frequency for compute-bound applications.

The main challenge is to generate an efficient and lightweight access phase to tackle memory bound applications, provided that the compiler cannot statically determine the right balance between the overhead of computing a memory address and the benefits obtained from prefetching the address. We approach this challenge using a static multi-versioning technique (*SMVDAE*), which incurs no runtime overhead in generating promising access versions, but provides flexibility and adaptability, as the system can explore a larger search space and can select the best performing access version dynamically.

We show that general purpose, complex applications not only exhibit energy improvements, but also significant performance boosts, when executed using the decoupled access-execute model, as decoupled execution exposes more memory and instruction level parallelism. While previous DAE compiler techniques were restricted to scientific task based parallel codes, SMVDAE targets: complex, irregular, general-purpose codes efficiently; yielding overall EDP improvements of 22%, with peak improvements of 72% for memory bound applications.

References

[1] SPEC CPU2006 function profile. `http://hpc.cs.tsinghua.edu.cn/research/cluster/SPEC2006Characterization/fprof.html`. Accessed: 2014-08-17.

[2] Transactional synchronization in haswell. `https://software.intel.com/en-us/blogs/2012/02/07/transactional-synchronization-in-haswell`.

[3] M. Arenaz, J. Tourio, and R. Doallo. An inspector-executor algorithm for irregular assignment parallelization. In J. Cao, L. Yang, M. Guo, and F. Lau, editors, *Parallel and Distributed Processing and Applications*, volume 3358 of *Lecture Notes in Computer Science*, pages 4–15. Springer Berlin Heidelberg, 2005.

[4] M. Arnold and B. G. Ryder. A framework for reducing the cost of instrumented code. In *ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, 2001.

[5] D. K. Chen, J. Torrellas, and P. C. Yew. An efficient algorithm for the run-time parallelization of doacross loops. In *Proceedings of the 1994 ACM/IEEE Conference on Supercomputing*, Supercomputing '94, pages 518–527, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press.

[6] J. Chen, H. Yi, X. Yang, and L. Qian. Compile-time energy optimization for parallel applications in on-chip multiprocessors. In V. Alexandrov, G. van Albada, P. Sloot, and J. Dongarra, editors, *Computational Science ICCS 2006*, volume 3992 of *Lecture Notes in Computer Science*, pages 904–911. Springer Berlin Heidelberg, 2006.

[7] X. Chen and S. Long. Adaptive multi-versioning for openmp parallelization via machine learning. In *Parallel and Distributed Systems (ICPADS), 2009 15th International Conference on*, pages 907–912, Dec 2009.

[8] T. Chilimbi and M. Hirzel. Dynamic hot data stream prefetching for general-purpose programs. In *ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, 2002.

[9] 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.

[10] B. Dufour, B. G. Ryder, and G. Sevitsky. Blended analysis for performance understanding of framework-based applications. In *Int'l Symp. on Software Testing and Analysis (ISSTA)*, 2007.

[11] S. Eyerman and L. Eeckhout. A counter architecture for online dvfs profitability estimation. *Computers, IEEE Transactions on*, 59(11):1576–1583, 2010.

[12] R. Gonzalez and M. Horowitz. Energy dissipation in general purpose microprocessors. *Solid-State Circuits, IEEE Journal of*, 31(9):1277–1284, Sep 1996.

[13] P. Hammarlund, A. J. Martinez, A. A. Bajwa, D. L. Hill, E. Hallnor, H. Jiang, M. Dixon, M. Derr, M. Hunsaker, R. Kumar, R. B. Osborne, R. Rajwar, R. Singhal, R. D'Sa, R. Chappell, S. Kaushik, S. Chennupaty, S. Jourdan, S. Gunther, T. Piazza, and T. Burton. Haswell: The fourth-generation intel core processor. *IEEE Micro*, 34(2):6–20, 2014.

[14] M. Hauswirth and T. M. Chilimbi. Low-overhead memory leak detection using adaptive statistical profiling. In *Int'l Conf. on Architectural Support for Programming Language and Operating Systems (ASPLOS)*, 2004.

[15] J. L. Henning. Spec cpu2006 benchmark descriptions. *SIGARCH Comput. Archit. News*, 34(4):1–17, Sept. 2006.

[16] M. Hirzel and T. Chilimbi. Bursty tracing: A framework for low-overhead temporal profiling. In *ACM Workshop on FeedbackDirected and Dynamic Optimization (FDD)*, 2001.

[17] Intel. Intel ® architecture instruction set extensions programming reference, pp.506-529, 2012.

[18] A. Jaleel. Memory characterization of workloads using instrumentation-driven simulation. *http://http://www.glue.umd.edu/~ajaleel/workload/*, 2007.

[19] A. Jimborean, P. Clauss, J.-F. Dollinger, V. Loechner, and J. M. Martinez Caamaño. Dynamic and speculative polyhedral parallelization using compiler-generated skeletons. *Int'l Journal of Parallel Programming (IJPP)*, 42(4):529–545, Aug. 2014.

[20] A. Jimborean, M. Herrmann, V. Loechner, and P. Clauss. Vmad: A virtual machine for advanced dynamic analysis of programs. In *IEEE Int'l Symp. on Performance Analysis of Systems and Software (ISPASS)*, pages 125–126, 2011.

[21] A. Jimborean, K. Koukos, V. Spiliopoulos, D. Black-Schaffer, and S. Kaxiras. Fix the code. don't tweak the hardware: A new compiler approach to voltage-frequency scaling. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '14, pages 262:262–262:272, New York, NY, USA, 2014. ACM.

[22] A. Jimborean, L. Mastrangelo, V. Loechner, and P. Clauss. Vmad: An advanced dynamic program analysis and instrumentation framework. In *IEEE / ACM International Conference on Compiler Construction (CC)*, pages 220–239, 2012.

[23] M. Kamruzzaman, S. Swanson, and D. M. Tullsen. Inter-core prefetching for multicore processors using migrating helper threads. In *Proceedings of the sixteenth international conference on Architectural support for programming languages and operating systems*, ASPLOS '11, pages 393–404, New York, NY, USA, 2011. ACM.

[24] S. Kaxiras and M. Martonosi. Computer architecture techniques for power-efficiency. *Synthesis Lectures on Computer Architecture*, 3(1):1–207, 2008.

[25] G. Keramidas, V. Spiliopoulos, and S. Kaxiras. Interval-based models for run-time dvfs orchestration in superscalar processors. In *Proceedings of the 7th ACM international conference on Computing frontiers*, CF '10, pages 287–296, New York, NY, USA, 2010. ACM.

[26] K. Koukos, D. Black-Schaffer, V. Spiliopoulos, and S. Kaxiras. Towards more efficient execution: A decoupled access-execute approach. In *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing*, ICS '13, pages 253–262, New York, NY, USA, 2013. ACM.

[27] M. A. Laurenzano, Y. Zhang, L. Tang, and J. Mars. Protean code: Achieving near-free online code transformations for warehouse scale computers. In *IEEE/ACM Int'l Symp. on Microarchitecture (MICRO)*, pages 558–570, 2014.

[28] The llvm compiler infrastructure. `http://llvm.org`.

[29] L. Luo, Y. Chen, C. Wu, S. Long, and G. Fursin. Finding representative sets of optimizations for adaptive multiversioning applications. In *International Workshop on Statistical and Machine learning approaches to ARchitectures and compilaTion*, Paphos, Cyprus, Jan. 2009.

[30] D. Marino, M. Musuvathi, and S. Narayanasamy. Literace: effective sampling for lightweight data-race detection. In *ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, 2009.

[31] R. Miftakhutdinov, E. Ebrahimi, and Y. N. Patt. Predicting performance impact of dvfs for realistic memory systems. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-45, pages 155–165, Washington, DC, USA, 2012. IEEE Computer Society.

[32] P. J. Mucci, S. Browne, C. Deane, and G. Ho. Papi: A portable interface to hardware performance counters. In *In Proceedings of the Department of Defense HPCMP Users Group Conference*, pages 7–10, 1999.

[33] T. K. Prakash and L. Peng. Performance characterization of SPEC CPU2006 benchmarks on Intel Core 2 Duo processor. *ISAST Trans. Comput. Softw. Eng.*, 2(1):36–41, 2008.

[34] C. G. Quiñones, C. Madriles, J. Sánchez, P. Marcuello, A. González, and D. M. Tullsen. Mitosis compiler: An infrastructure for speculative threading based on pre-computation slices. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, pages 269–279, New York, NY, USA, 2005. ACM.

[35] L. Rauchwerger, N. M. Amato, and D. A. Padua. Run-time methods for parallelizing partially parallel loops. In *Proceedings of the 9th International Conference on Supercomputing*, ICS '95, pages 137–146, New York, NY, USA, 1995. ACM.

[36] J. Saltz, R. Mirchandaney, and K. Crowley. Run-time parallelization and scheduling of loops. *Computers, IEEE Transactions on*, 40(5):603–612, May 1991.

[37] J. E. Smith. Decoupled access/execute computer architectures. *SIGARCH Comput. Archit. News*, 10(3):112–119, Apr. 1982.

[38] V. Spiliopoulos, S. Kaxiras, and G. Keramidas. Green governors: A framework for continuously adaptive dvfs. In *Proceedings of the 2011 International Green Computing Conference and Workshops*, IGCC '11, pages 1–8, Washington, DC, USA, 2011. IEEE Computer Society.

[39] J. A. Stratton, C. Rodrigues, I.-J. Sung, N. Obeid, L.-W. Chang, N. Anssari, G. D. Liu, and W.-M. W. Hwu. Parboil: A revised benchmark suite for scientific and commercial throughput computing. *Center for Reliable and High-Performance Computing*, 2012.

[40] S. Tavarageri and P. Sadayappan. A compiler analysis to determine useful cache size for energy efficiency. In *Int'l Parallel and Distributed Processing Symp. Workshops (IPDPSW)*, pages 923–930, 2013.

[41] K. Tian, Y. Jiang, E. Z. Zhang, and X. Shen. An input-centric paradigm for program dynamic optimizations. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '10, pages 125–139, New York, NY, USA, 2010. ACM.

[42] F. Xie, M. Martonosi, and S. Malik. Compile-time dynamic voltage scaling settings: Opportunities and limits. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, PLDI '03, pages 49–62, New York, NY, USA, 2003. ACM.

[43] D. Yokota, S. Chiba, and K. Itano. A new optimization technique for the inspector-executor method. In *IASTED PDCS*, pages 706–711, 2002.

[44] T. Yuki and S. Rajopadhye. Folklore confirmed: Compiling for speed = compiling for energy. In C. Cacaval and P. Montesinos, editors, *Languages and Compilers for Parallel Computing*, volume 8664 of *Lecture Notes in Computer Science*, pages 169–184. Springer International Publishing, 2014.

[45] W. Zhang, D. Tullsen, and B. Calder. Accelerating and adapting precomputation threads for effcient prefetching. In *High Performance Computer Architecture, 2007. HPCA 2007. IEEE 13th International Symposium on*, pages 85–95, Feb 2007.

[46] M. Zhou, X. Shen, Y. Gao, and G. Yiu. Space-efficient multiversioning for input-adaptive feedback-driven program optimizations. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '14, pages 763–776, New York, NY, USA, 2014. ACM.