Compiler Support for Selective Page Migration in NUMA Architectures

Guilherme Piccoli
Unicamp
guilherme.piccoli@students.ic.unicamp.br

Henrique N. Santos
UFMG
hnsantos@dcc.ufmg.br

Raphael E. Rodrigues
UFMG
raphael@dcc.ufmg.br

Christiane Pousa
ETH Zürich
pousa@id.ethz.ch

Edson Borin
Unicamp
edson@ic.unicamp.br

Fernando Magno Quintão Pereira
UFMG
fernando@dcc.ufmg.br

ABSTRACT

Current high-performance multicore processors provide users with a non-uniform memory access model (NUMA). These systems perform better when threads access data on memory banks next to the core where they run. However, ensuring data locality is difficult. In this paper, we propose compiler analyses and code generation methods to support a lightweight runtime system that dynamically migrates memory pages to improve data locality. Our technique combines static and dynamic analyses and is capable of identifying the most promising pages to migrate. Statically, we infer the size of arrays, plus the amount of reuse of each memory access instruction in a program. These estimates rely on a simple, yet accurate, trip count predictor of our own design. This knowledge let's us build templates of dynamic checks, to be filled with values known only at runtime. These checks determine when it is profitable to migrate data closer to the processors where this data is used. Our static analyses are quadratic on the number of variables in a program, and the dynamic checks are $O(1)$ in practice. Our technique does not require any form of user intervention, neither the support of a third-party middleware, nor modifications in the operating system's kernel. We have applied our technique on several parallel algorithms, which are completely oblivious to the asymmetric memory topology, and have observed speedups of up to 4x, compared to static heuristics. We compare our approach against Minas, a middleware that supports NUMA-aware data allocation, and show that we can outperform it by up to 50% in some cases.

Categories and Subject Descriptors

D.3.4 [**Software**]: Programming Languages—*Compilers*

PACT'14, August 24–27, 2014, Edmonton, AB, Canada.
Copyright 2014 ACM 978-1-4503-2809-8/14/08 ...$15.00.
http://dx.doi.org/10.1145/2628071.2628077 .

General Terms

Languages

Keywords

NUMA, static analysis, data placement, memory topology

1. INTRODUCTION

There exist two basic ways to connect processors to memory in a multi-core architecture. The simplest alternative consists in connecting each processor equally to the memory bank. Such architectures are also known as *Symmetric Multiprocessing* (SMP) systems. The other approach is asymmetric: each processor is directly connected to local memory banks and a forwarding mechanism is used to read or write data in remote memory banks. This kind of architecture, known as *Non-Uniform Memory Access*, or NUMA for short, is not a new idea - early designs were already enjoying commercial success in the late 70's. As an example, the Burroughs's B6800 processor, of 1977, is a NUMA architecture. Although the NUMA model used to be mostly common in high performance computing systems, nowadays, we find non-uniform memory layouts in several general-purpose multi-core processors such as the Tilera Tile64, Intel E8870, HP sx2000, and the AMD Opteron. This tendency is likely to remain in vogue, given that the number of cores per chip is predicted to increase even more in the coming years [2, Sec.2.2]. The attractiveness of the NUMA design comes from the fact that it enables parallel data accesses on multiple memory banks; thus reducing memory access conflicts. The ever-increasing speed of processing units is contributing to boost the importance of this advantage.

The performance of memory systems, in general, suffers in face of access conflicts. Such conflicts happen whenever different threads try to read or write data in the same memory bank simultaneously. NUMA machines mitigate this problem by allowing the placement of data in separate memory banks. Therefore, in these architectures, data allocation plays a central role in performance. Ideally, a thread should access data placed in memory located at the core that executes it. We call this memory a local bank, and we call these accesses *local.* Whenever a thread reads or writes data in a remote (non-local) memory bank, the data is copied from a remote location to a local cache - an expensive operation that we call a *remote access*. Yet, the main drawback of remote accesses is not their long traffic time: these accesses are prone to generate access conflicts, leading to resource contention between threads [27].

There have been several different proposals to improve data allocation on NUMA machines. Many of these techniques were published in the late 2000's. Some of these methods are implemented at the operating system level [2, 4, 17, 23], at the hardware level [10, 15, 28], or at the middleware level [6, 8, 19, 24]. These approaches extract information from a program's runtime behavior, but do not take its coding structure into consideration. Other approaches rely on library calls, which give the programmer the opportunity to move memory pages around [5, 14, 26, 29]. These techniques, in our point of view, have a shortcoming: they require the intervention of the programmer. Data placement libraries, for instance, provide developers with ways to determine the allocation of data before a program starts running, or let them migrate memory pages during the program's execution. To benefit from this support, users must be familiar with the underlying memory architecture and with the application's code.

In this paper we have designed and implemented a compiler assisted technique that solves these shortcomings. We introduce a suite of compiler analyses that determine when it is beneficial to move data once a program starts executing. We also use our analyses to generate dynamic checks, which we insert into the program's code. These checks are built as templates that once filled with runtime values - mostly the limits of loops - decide if the operating system should migrate physical memory pages to the cores where they are frequently accessed. The proposed approach is very granular: it can treat data used in different parts of the program in different ways, selectively moving only the memory pages whose migration pays off. Our solution does not demand any extra hardware support. Furthermore, it requires minimal support from the operating system. More specifically, we need the ability to move pages around and some limited form of introspection to find the size of the cache. This support is already present in standard distributions of popular operating systems. Most importantly, our approach does not require any form of user intervention in the application code.

We have implemented our technique using the LLVM [21] compiler, and have tested it on a number of parallel benchmarks, which we ship together with our tool. These benchmarks have been coded using C POSIX Threads and are completely oblivious to the topology of the memory hierarchy. Our results show that programs instrumented with our code transformations outperform the original parallel programs by up to four times, on a 64-core NUMA machine. Furthermore, we compare our approach against Minas [26], a middleware that optimizes data allocation and placement in non-uniform memory access architectures. Minas provides developers with an API which gives them (i) information about the topology of the machine and (ii) mechanisms to determine the initial allocation and placement of data into memory banks. The programs that we generate automatically tend to outperform the programs hand-coded to use the Minas library. These results are encouraging, because our approach does not require any form of intervention from the programmer, whereas Minas is used as a library, which must be manipulated directly by the developer.

2. BACKGROUND

In order to take more benefit from the ever-increasing core count, computer architects have been shifting their designs from the SMP (Symmetric Multiprocessing) model towards the NUMA (Non-Uniform Memory Access) design [20]. Similar to the SMP approach, NUMA architectures allow code running in any core to access any memory word in the system. However, in the NUMA model, the memory banks are distributed among multiple controllers and the data access latency varies accordingly to the memory bank and the core position. Figure 1 shows the topology of a NUMA architecture with four processing cores and four memory controllers. Each memory controller is connected to a memory bank, to a computing core, and to the other memory controllers through the interconnecting bus.

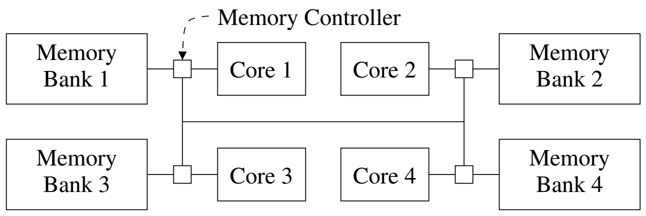


Figure 1: Non-Uniform Memory Access system.

On a NUMA machine, threads running on different cores can access data on different memory banks. However, to develop applications that take full benefit from this capacity, it is important to distribute data on memory banks located next to the cores where the threads that access them are executed. In order to induce this locality, modern operating systems try to allocate pages on memory banks next to the cores that are executing the threads that triggered the page allocation. This is known as "first-touch" policy. However, in many programs, the key data structures are allocated in the main thread, which runs before parallel computation takes place. This setup causes all the data to be allocated on a single memory bank. Unfortunately, in this scenario, all the cores will compete for the same memory controller, as in the SMP approach. The "first-touch" curve in Figure 2 shows the scalability of a parallel program that performs the Cholesky decomposition on several matrices that were initially allocated in the main thread.

A common optimization used in this environment consists in allocating or moving memory pages to memory banks next to the cores executing the threads that tend to use these pages more often [2, 20]. Figure 2 illustrates the remarkable benefits of this technique. The Linux kernel provides system calls to migrate memory pages between memory banks. After migrating the pages, the system updates the page tables to map the virtual page addresses to the new physical page addresses. However, migrating the page may be an expensive operation, and the optimization may only pay off if the data in the page is frequently retrieved from main memory [17]. In this work, we leverage static compiler optimizations and runtime support to improve performance via selective page migration. The gains in Figure 2 were obtained via manual intervention in the source code. We want to derive such gains automatically, via transformations implemented at the compiler level.

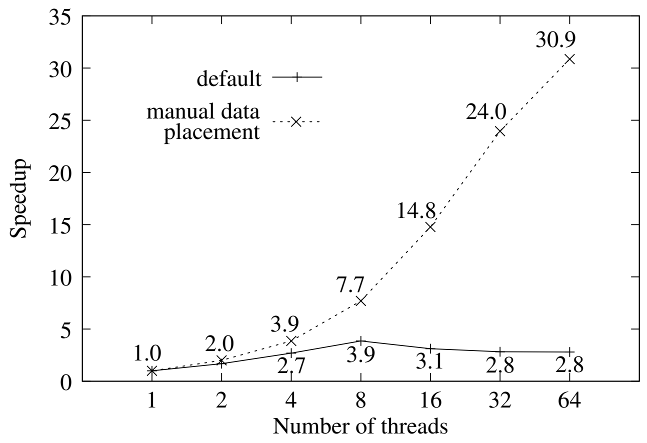


Figure 2: Scalability of a parallel program with first-touch and manual page placement on a 64-core NUMA system. Numbers give speedup compared to a single-threaded application.

3. SELECTIVE PAGE MIGRATION

Our approach consists in moving pages that we predict to be frequently reused (hot pages) to local memory banks. Most of the page accesses occur inside program loops; hence, our compiler inserts checks before loops that contain array accesses. Such checks determine if it is profitable to migrate the page before the program's flow of execution enters these loops.

Figure 3 illustrates our approach. Throughout the rest of this section, we will use the program in Figure 3 (a), an artificial code that exercises all the facets of the technique proposed here, as our running example. The pattern of loops and array accesses in this program will let us illustrate the main features of our static analyses. We will represent every array access as $a[e]$, where a is a base pointer, and e is an indexing expression. In other words, we linearize every array access. If a is an $N \times M$ matrix, then we write $a[i, j]$ as $a[i \times M + j]$. This notation puts us closer to the internal representation of arrays in our compiler infrastructure. In this example, `Tid` is a special variable, which represents the thread identifier. Figure 3 (b) shows the instrumentation that we insert. We always try to insert code surrounding outermost loops. The conditional test $(\texttt{Tid} + 1) \times R + P > \texttt{Tid} \times R$ in Figure 3 (b) guards us against scenarios where the outermost loop, at line 1, executes zero iterations. With this, we avoid burdening applications that would not benefit from page migration with an unnecessary overhead.

We consider that a block of data should be transferred to a local memory bank if: (i) its memory footprint is larger than the processor's last level cache, and (ii) it is accessed several times by the local thread. We take item (i) - cache size - into consideration because if the data fits in the cache, then it will be naturally copied while accessed. Item (ii) - reuse - is important because the cost of moving a page over is roughly equivalent to the cost of accessing each of its elements once. Thus, we do not save memory accesses by migrating a page whose data is only read or written a few times. We take care of item (i) by determining the size of the arrays that the program manipulates. To determine (ii), we discover the number of times that each array is accessed

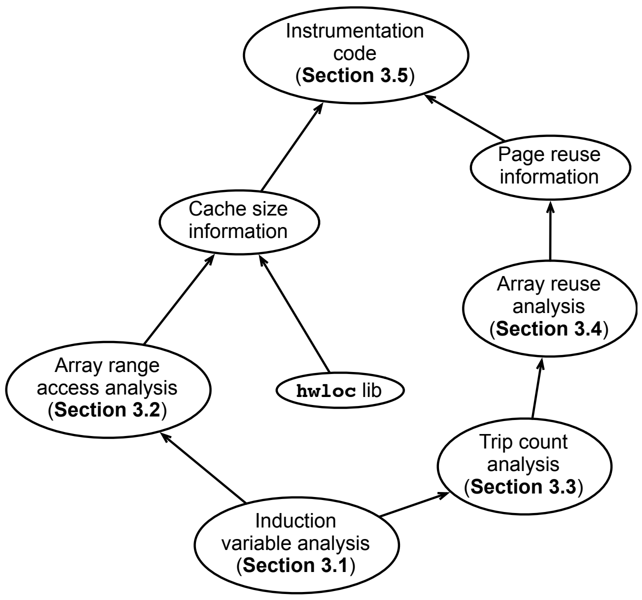
(a)
```
1  for (i = Tid × R; i < (Tid + 1) × R + P; i++) {
2    A[i] = 0;
3    for (j0 = i + 1; j0 < N2; j0 += 4) {
4      B[i×N2 + j0] = 0;
5      for (k = j0; k > i; k -= 1) {
6        B[i×N2 + j0] = B[i×N2 + j0] + C[i×N2×N3 + j0×N3 + k];
7      }
8    }
9    for (j1 = i + 1; j1 < N2; j1 += 4) {
10     A[i] = A[i] + B[i×N2 + j1];
11   }
12 }
```

(b)
```
 if ((Tid + 1) × R + P > Tid × R) {
   (ArraySt_A, ArrayEd_A) = calc_size_exp_A(N1)
   (ArraySt_B, ArrayEd_B) = calc_size_exp_B(N1, N2)
   (ArraySt_C, ArrayEd_C) = calc_size_exp_A(N1, N2, N3)

   ArrayReuse_A = calc_reuse_exp_A(N1)
   ArrayReuse_B = calc_reuse_exp_B(N1, N2)
   ArrayReuse_C = calc_reuse_exp_C(N1, N2, N3)

   PageReuse_A = ArrayReuse_A / ArraySize_A × PageSize
   PageReuse_B = ArrayReuse_B / ArraySize_B × PageSize
   PageReuse_C = ArrayReuse_C / ArraySize_C × PageSize

   try_migrate(A, ArraySt_A, ArrayEd_A, PageReuse_A)
   try_migrate(B, ArraySt_B, ArrayEd_B, PageReuse_B)
   try_migrate(C, ArraySt_C, ArrayEd_C, PageReuse_C)
 }
1  for (i = Tid × R; i < (Tid + 1) × R + P; i++) {
...  ...
12 }
 release_migrated_pages(A, ArraySt_A, ArrayEd_A)
 release_migrated_pages(A, ArraySt_A, ArrayEd_A)
 release_migrated_pages(A, ArraySt_A, ArrayEd_A)
```

Figure 3: (a) Our core example. (b) Instrumented program.

within a loop. We call such metric the *reuse of the array.* As the reader will notice in Figure 3 (b), we ultimately consider reuse as an estimate of how often the same page is accessed (see the assignment to `PageReuse_A`, for instance). This happens because our unit of data migration is the size of the operating system's virtual page.

In the rest of this section, we describe the static analyses that we use to obtain all the information that our heuristics require. Figure 4 summarizes them. In Section 3.2 we show how to compute symbolic expressions denoting the least and maximum addresses that an array can assume. In order to build these expressions we need one key information: the ranges, i.e., minimum and maximum symbolic values that the variables used to index arrays can assume. We obtain this information via the induction variable analysis that we describe in Section 3.1. In Section 3.4 we discuss how we build symbolic expressions representing the reuse of arrays. To this end, we need to construct symbolic expressions that



Instrumentation code (Section 3.5)
Page reuse information
Cache size information
Array reuse analysis (Section 3.4)
Array range access analysis (Section 3.2)
hwloc lib
Trip count analysis (Section 3.3)
Induction variable analysis (Section 3.1)

Figure 4: A bird's-eye view of our approach.

describe how many times a nest of loops will execute; we accomplish this task through the analysis discussed in Section 3.3. These symbolic expressions give us the subsidies to instrument the code to perform dynamic tests, as we describe in Section 3.5. These tests require information concerning the topology of the underlying memory architecture, which we obtain via the Portable Hardware Locality (`hwloc`) library [7], again, without any intervention of the programmer.

Loop Jargon. Let S be a subset of nodes of a control flow graph G. S contains a special node H, which we shall call *header* or *entry point*. Following Appel and Palsberg [1, pp.376], we say that S is a *natural loop* if and only if it presents the following three properties:

1. there exists a path from any node in S to H;
2. there exists a path from H to any node in S;
3. there is no path from any node of G to any node of S that does not go across H.

The last property defines S as a *single-entry* region, following Ferrante's nomenclature [16]. An edge between any node in S to H is called a *back-edge*. We adopt Wolfe's definition of *trip count* [31, pp.200]: the number of times any back-edge of a natural loop has been traversed by the program flow within a single execution of the loop.

We call a node $L \in S$ a *latch* or *exit point* if there exists an edge from L to a node N, $N \in G$, $N \notin S$. We say that L is a *natural latch* if one of these two conditions applies:

- $L = H$. In this case we have a *while loop*;
- $L \neq H$, and, for any edge e that starts at L, either e leaves S, or e goes to H. In this case we have a *repeat loop*.

If S contains only one latch, then we call it *single exit*. In this work we consider multiple exit loops featuring only one natural latch. Code generated from typical programming language constructs, i.e., `for`, `while` and `repeat` has this property, as long as the command `goto` is not used.

Symbolic Kernels. In this work we have designed static analyses that associate variables and program labels with *symbols*. We define the *symbolic kernel* of a loop S with header H as the set of variables *alive* at H that are not defined inside S plus the constants (known at compilation time) that are used in the body of S. We adopt the standard definition of *liveness*, i.e., we say that a variable v is alive at a node n of a control flow graph if:

- there exists a path from a point n_d, where v is defined, to a point n_u, where v is used, that goes across n;
- no other definition of v exists between n_d and n.

As an example, variables `Tid`, R, P, N_2, and N_3 constitute the symbolic kernel of the natural loop that starts in line 1 of Figure 3 (a). Similarly, variables `Tid`, R, P, N_2, N_3, i and j_0 are the symbolic kernel of the loop at line 5 in the same figure. We say that E is an *expression* if and only if E is defined by the grammar below. In this definition, s is a symbol, i.e., an element in the symbolic kernel and $n \in \mathbb{N}$:

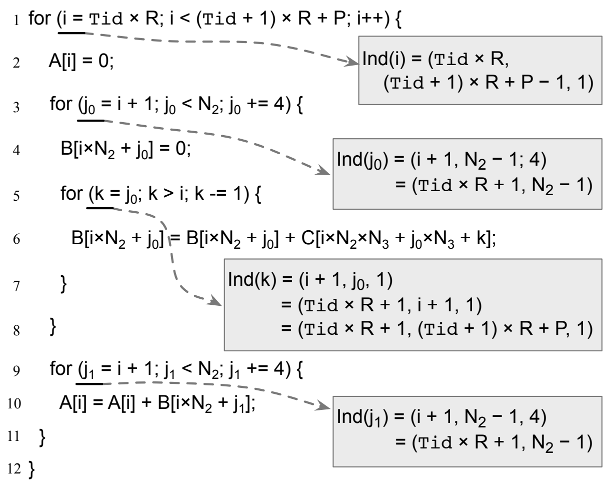
$$E \;::=\; n \mid s \mid \min(E,E) \mid \max(E,E) \mid E-E \mid E+E \mid E\times E \mid E/E$$

The techniques that we describe in this paper consist in generating - statically - expressions representing the size and reuse of arrays. We solve these expressions dynamically, once we replace symbols with their runtime values.

3.1 Induction Variable Analysis

The induction variable analysis associates each loop induction variable v with a triple $Ind(v) = (l, u, s)$. The element l is a symbolic expression representing the minimum value that v might assume during the execution of the loop. The parameter u represents the maximum value that v may assume, and $s \in \mathbb{N}$ represents the step of change of v. In this work we restrict our analyses to loops that are controlled by *linearly monotonic induction variables*. The variable `i` is a linear induction variable in a loop if the only definitions of `i` within that loop are of the form `i` = `i` + `c` or `i` = `i` − `c`, where `c` is loop invariant. We call `c` the *step* of `i`. If every redefinition of `i` uses invariants with the same signal, then we say that `i` is monotonic. Any latch contains a *stop condition*: a boolean expression whose evaluation either keeps the program flow in S or leads away from it. If the stop condition uses an induction variable `i`, then we say that the loop is controlled by `i`. For instance, the loop `for(k = j0; k > i; k -= 1)`, at line 5 of Figure 3 (a), has a stop condition `k > i`, which is controlled by variables `i` and `k`. The step of induction variable `k` is one in this example.

To discover the minimum and maximum values that linear monotonic induction variables can assume, we look at the code that initializes them and at the code that limits them. If we have a sequence of nested loops $L_0 \ldots L_n$, where L_i is nested within L_{i-1}, then we run our analysis in L_{i-1} before visiting L_i. With this, we try to build the limits of the induction variable used in L_i as a function of the limits of the induction variables used in L_{i-1}. Figure 5 illustrates our analysis. In this example, first we analyze the loop at line 1 of the example program. This gives us limits for i, the only induction variable used in that loop. Once we find symbolic bounds of i, we move on to discover the limits of



```
1  for (i = Tid × R; i < (Tid + 1) × R + P; i++) {
2    A[i] = 0;
3    for (j0 = i + 1; j0 < N2; j0 += 4) {
4      B[i×N2 + j0] = 0;
5      for (k = j0; k > i; k -= 1) {
6        B[i×N2 + j0] = B[i×N2 + j0] + C[i×N2×N3 + j0×N3 + k];
7      }
8    }
9    for (j1 = i + 1; j1 < N2; j1 += 4) {
10     A[i] = A[i] + B[i×N2 + j1];
11   }
12 }
```

```
Ind(i) = (Tid × R,
   (Tid + 1) × R + P − 1, 1)

Ind(j0) = (i + 1, N2 − 1; 4)
        = (Tid × R + 1, N2 − 1)

Ind(k) = (i + 1, j0, 1)
       = (Tid × R + 1, i + 1, 1)
       = (Tid × R + 1, (Tid + 1) × R + P, 1)

Ind(j1) = (i + 1, N2 − 1, 4)
        = (Tid × R + 1, N2 − 1)
```

Figure 5: Induction variable analysis applied on the program seen in Figure 3 (a).

j_0 and j_1. These variables control the loops at lines 3 and 9, which are nested within the loop on line 1. Therefore, the limits of these two induction variables are built as functions of the limits of i. Finally, we place bounds on k. These bounds are built as functions of the bounds of j_0 and i. To simplify the formulas that we build in such fashion, we use the SymPy library [18] of symbolic manipulation. SymPy uses cylindrical algebraic decomposition [3] to solve the symbolic equations that we feed to it.

Dealing with Multi-Path Induction Variables. An induction variable can be incremented with different steps on the same loop. This happens, for instance, in the program that we show in Figure 6 (Left). In this paper we are interested in the minimum trip count of a loop. In other words, we want to estimate this quantity conservatively. Therefore, we shall be considering the maximum increment over all paths through which an induction variable can be incremented. We can compute this number via Dijkstra's algorithm, which solves the single-source shortest path problem for a graph with non-negative edge path costs [13]. We apply the algorithm on the program's dependence graph, a data structure due to Ferrante *et al.* [16]. In our case, the cost of each edge is the increment that the induction variable can suffer along that path.

Figure 6 (Right) shows the program dependence graph of our example. We use Static Single Assignment form (SSA) [11] to represent programs. In this representation, each variable of a program has only one definition site. Therefore, each redefinition of variable i in Figure 6 (Left) has a new name in the graph in Figure 6 (Right). The ϕ-functions in Figure 6 (Right) join different definitions of variables. As an example, the increment at line 9 can be applied on three different definitions of i. These definitions reach line 9 coming from lines 4, 6, and 8. All these definitions are unified into $\mathtt{i}_5$, which is then incremented; thus producing a new name $\mathtt{i}_6$.

Dijkstra gives us, in this example, the heaviest path (+4, +1) = +5, where the weight of a path is the sum of all the increments that the induction variable can suffer along it. We cannot deal with *oscillating* induction variables, i.e., variables that can suffer positive or negative increments through an iteration of a loop. Nevertheless, this situation is rather uncommon, and we have not observed it in a suite of hundreds of benchmarks, as we will discuss in Section 4.

```
1  i = 0
2  while (i < N) {
3    if (DEF_INT_32)
4      i += 4;
5    else if (DEF_INT_16)
6      i += 2;
7    else
8      i += 1;
9    i++;
10 }
```

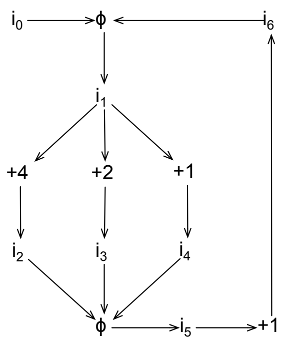


Figure 6: Example with multiple paths in a single loop.

3.2 Array Range Access Analysis

If $a[e]$ is an array access, then we say that the *access range* of $a[e]$ is the minimum and the maximum values that e might assume. We find these two values, i.e., the symbolic minimum and maximum of e, in a two steps approach:

1. we replace every induction variable v used in e by the interval $[l, u]$, where l and u are given by $\mathit{Ind}(v) = (l, u, s)$. Let the resulting symbolic expression be denoted by e_i.

2. we solve e_i using a simple algebra of symbolic intervals, which is defined by the expressions below:

   - $[l_0, u_0] + [l_1, u_1] = [l_0 + l_1, u_0 + u_1]$
   - $[l_0, u_0] - [l_1, u_1] = [\min(l_0 - l_1, u_0 - u_1), \max(l_0 - l_1, u_0 - u_1)]$
   - $[l_0, u_0] \times [l_1, u_1] = [\min(T), \max(T)]$, where $T = \{l_0 \times l_1, l_0 \times u_1, u_0 \times l_1, u_0 \times u_1\}$
   - $[l_0, u_0]/[l_1, u_1] = [\min(T), \max(T)]$, where $T = \{l_0/l_1,\ l_0/u_1, u_0/l_1, u_0/u_1\}$, if $0 \notin [l_1, u_1]$, otherwise $[-\infty, +\infty]$

Continuing with our example, Figure 7 shows the results of this analysis for three of the array accesses seen in the program of Figure 3 (a). Once we solve the interval expression e_i, for an array access $a[e]$, we bind this access to a symbolic interval $[l, u]$. The syntax $A[i]$ denotes the accesses at lines 2 and 10 of Figure 3. $B[i \times N_2 + j_0]$ denotes the accesses at lines 4 and 6. The access on C happens at line 6 of Figure 3. Notice that internally we use the static single assignment form. Thus, each induction variable has a unique name. Consequently, if a program contains two accesses using the same expression, e.g., $A[e]$ and $B[e]$, then these accesses will give us the same ranges. Also notice that our interval algebra admits several simplifications, mostly concerning products between intervals. For instance, the product $[l, u] \times n, n \in \mathbb{N}, n > 0$ is given by $[n \times l, n \times u]$. As in Section 3.1, we rely on SymPy to perform algebraic simplifications on symbolic expressions.

$$A[i] \quad \begin{cases} min = \texttt{Tid} \times R \\ max = \texttt{Tid} \times R + P - 1 \end{cases}$$

$$B[i \times N_2 + j_0] \quad \begin{cases} min = \texttt{Tid} \times R \times N_2 + 1 \\ max = (\texttt{Tid} \times R + P) \times N_2 - 1 \end{cases}$$

$$C[i{\times}N_2{\times}N_3{+}j_0{\times}N_3{+}k] \begin{cases} min = \texttt{Tid} \times R \times N_2 \times N_3 + \\ \qquad (\texttt{Tid} \times R + 1) \times (N_3 + 1) \\ max = (((\texttt{Tid} + 1) \times R + P) \times N_2 - 1) \\ \qquad \times N_3 + N_2 - 1 \end{cases}$$

Figure 7: Results of the symbolic array range access analysis.

From ranges to sizes. The array range access analysis gives us a way to determine the lowest and highest addressable offsets in an array. Its lowest addressable offset is the minimum between all the lower ranges inferred for it. Naturally, its highest addressable offset is the maximum between all the upper ranges inferred for it. We estimate the size of an array as the difference between its highest and lowest addressable offsets. In Figure 3 (b), this number, for array A, is given by $\texttt{ArrayEd}_\texttt{A} - \texttt{ArraySt}_\texttt{A}$.

3.3 Trip Count Analysis

The reuse of an array a in a loop L is the number of times that a is accessed within L. Therefore, to estimate the reuse of an array a, we must estimate the number of iterations of every loop that contains accesses to a. We perform such estimates via trip count analysis. This analysis associates each loop L in the program with a summation, which gives us the number of times that L executes in relation to its immediate parent loop. If L is not nested within any other loop, then this summation gives us the number of times L executes in the parent function. If L is controlled by an induction variable v, then this summation is given by the expression:

$$Trip(L_v) = \sum_{i=l}^{u} \frac{1}{s}, \text{ where } Ind(v) = (l, u, s)$$

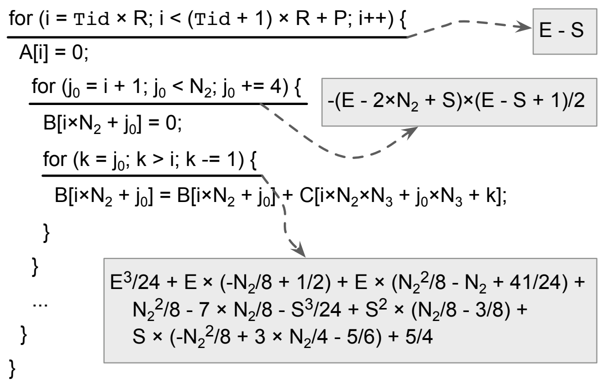
The number of times a loop L_n executes in relation to its parent loops (outermost to innermost) $L_1, L_2, \cdots, L_{n-1}$ is given by sequential application of each loop summation:

$$\sum_{i_1=l_1}^{u_1} \left(\frac{1}{s_1} \times \sum_{i_2=l_2}^{u_2} \left(\frac{1}{s_2} \times \cdots \left(\frac{1}{s_{n-1}} \times \sum_{i_n=l_n}^{u_n} \frac{1}{s_n} \right) \cdots \right) \right)$$

Figure 8 shows the results of our trip count analysis when applied on three loops of the program first seen in Figure 3 (a). All the summations are built as functions of the symbolic kernels of their respective loops. Our summations admit closed formulas, which we determine as successive applications of the identity:

$$\sum_{i=l}^{u} \frac{1}{s} = \frac{u + 1 - l}{s}$$

We obtain such formulas through the SymPy library. The three symbolic expressions seen in Figure 8 were produced



```
for (i = Tid × R; i < (Tid + 1) × R + P; i++) {
  A[i] = 0;
  for (j0 = i + 1; j0 < N2; j0 += 4) {
    B[i×N2 + j0] = 0;
    for (k = j0; k > i; k -= 1) {
      B[i×N2 + j0] = B[i×N2 + j0] + C[i×N2×N3 + j0×N3 + k];
    }
  }
  ...
 }
}
```

$E - S$
$-(E - 2{\times}N_2 + S){\times}(E - S + 1)/2$
$E^3/24 + E \times (-N_2/8 + 1/2) + E \times (N_2^2/8 - N_2 + 41/24) + N_2^2/8 - 7 \times N_2/8 - S^3/24 + S^2 \times (N_2/8 - 3/8) + S \times (-N_2^2/8 + 3 \times N_2/4 - 5/6) + 5/4$

Figure 8: Results of the trip count analysis. We let $S = \texttt{Tid} \times R$, and $E = (\texttt{Tid} + 1) \times R + P$.

in this way. Although they seem complicated, much of their complexity will be optimized away, once we run them through the compiler's standard optimization passes.

3.4 Array Reuse Analysis

Once we have the trip count of every loop L_i, we proceed to determine the reuse of each array. If a program contains an array access $a[e]$, then we say that this access is *immediately within* loop L_i if: (i) the instructions that implement $a[e]$ are inside L_i; and (ii) for every other loop L_j that contains $a[e]$, L_j contains L_i. For each array access, we say that the reuse of that access is the number of times that the access will be executed. We estimate the number of executions of an access as the trip count of its immediate loop. Given an array a, its reuse is the sum of the reuse of all its accesses in a loop nest. For instance, the reuse of array B in our example of Figure 3 is $Trip(L_3) + 2 \times Trip(L_5) + Trip(L_9)$, where L_i is the loop at line i in the figure. We use this formulate because array B is used once immediately within loops L_3 (line 4) and L_9 (line 10), and twice within loop L_5 (line 6). When generating final formulas for reuse estimation, we multiply the reuse of each array access by the page size, as this is the granularity of data that we can migrate.

3.5 Instrumentation Code

Figure 9 shows the final expressions that we insert in the original example to implement our heuristics. The minimum and maximum indexable offsets of the array are computed as the minimum among all the lower bounds and the maximum among all the upper bounds that the range access analysis finds. In the figure, we call these values `ArraySt_A` and `ArrayEd_A` respectively. As mentioned before, the array reuse is the sum of the reuse computed for each use of the array within the loop. In Figure 9 we show the closed formulas that we use to calculate the reuse of array A.

An important question that we must address is: where to place the instrumentation code? Ideally we would like to remove instrumentation checks from inner loops, to decrease the overhead on the target program. Therefore, we place these checks at the earliest program point where all the variables used in the symbolic kernel of a loop are alive. In the example of Figure 3 (a), this point is right before line

```
if ((Tid + 1) × R + P > Tid × R) {
  (ArraySt_A, ArrayEd_A) = (Tid × R,  (Tid + 1) × R + P − 1))
  (ArraySt_B, ArrayEd_B) = ...
  (ArraySt_C, ArrayEd_C) = ...

  ArrayReuse_A = (2 × N2 − 2 × Tid × R + R + P) × 2 × (R + P + 1)
  ArrayReuse_B = ...
  ArrayReuse_C = ...

  PageReuse_A = ArrayReuse_A / ArraySize_A × PageSize
  PageReuse_B = ...
  PageReuse_C = ...

  try_migrate(A, ArraySt_A, ArrayEd_A, PageReuse_A)
  ...
}
```

Figure 9: Final expressions used to estimate the size and reuse of array A, seen in Figure 3.

```
(a) try_migrate (Array, Start, End, PageReuse) {
      if (End - Start > CacheSize && PageReuse > Threshold ) {
        PageStart = (Array + Start) / PAGE_SIZE;
        PageEnd   = (Array + End) / PAGE_SIZE;
        (MinMigratablePageStart, MaxMigratablePageEnd) =
                                  pin_pages(gettid(), PageStart, PageEnd);
        if (MaxMigratablePageEnd - MinMigratablePageStart > 0) {
          migrate (PageStart, PageEnd);
        }
      }
    }

(b) release_migrated_pages (Array, Start, End) {
      PageStart = (Array + Start) / PAGE_SIZE;
      PageEnd   = (Array + End) / PAGE_SIZE;
      unpin_pages (PageStart, PageEnd);
    }
```

Figure 10: Functions used to implement page migration.

1 of the program. Notice that there exists always a point where we can place the dynamic checks: immediately before the header of the loop, because, by definition, all the variables in the symbolic kernel of the loop are alive at that point.

Dealing with disputes among threads. We implement page migration through two functions: `try_migrate` and `release_migrated_pages`. We show a high-level implementation of these functions at Figure 10. We define a *pinned page* as a page that has been migrated by a call to `try_migrate`. While a page remains in this state, we do not allow further migration of it. With this, we prevent threads from *disputing* a page. We say that two threads dispute a page if: (i) these threads run in different cores; and (ii) these threads try to migrate the same page. If threads were allowed to dispute pages, then the excessive copy of data between memory banks would impose a burden on our system.

Every memory page is initially *unpinned*. During the execution of the program, the state of a page can alternate between unpinned and *pinned*. If a thread touches a page p and successfully migrates it, then it marks p as pinned. Once the thread exits the loop where p is used, it turns that page's state back into unpinned. Our compiler inserts instrumentation code to operate these changes. In Figure 3 (b), page states can change at two points in the code of the instrumented program. Pages can be pinned by one of the three calls of `try_migrate`, right before the loop. In the opposite direction, pages might be unpinned by the three calls of `release_migrated_pages`, which occur right after the loop.

To discover which pages are pinned, we use a binary tree of interval ranges. Each node in this tree represents an interval of virtual memory addresses, which is associated with a thread identifier. The `try_migrate` function attempts to perform the migration of a contiguous range of pages, which is specified by the interval `PageStart - PageEnd`. It is possible that some of the pages in this range are pinned. In this case, the auxiliary function `pin_pages` scans our interval tree, and returns the first contiguous sequence of unpinned pages. Notice that we do not return a maximal sequence of unpinned pages, nor we return multiple intervals of unpinned pages, in case they exist. We use a best effort strategy because we have realized experimentally that this approach yields a good tradeoff between the precision of our heuristic and its efficiency.

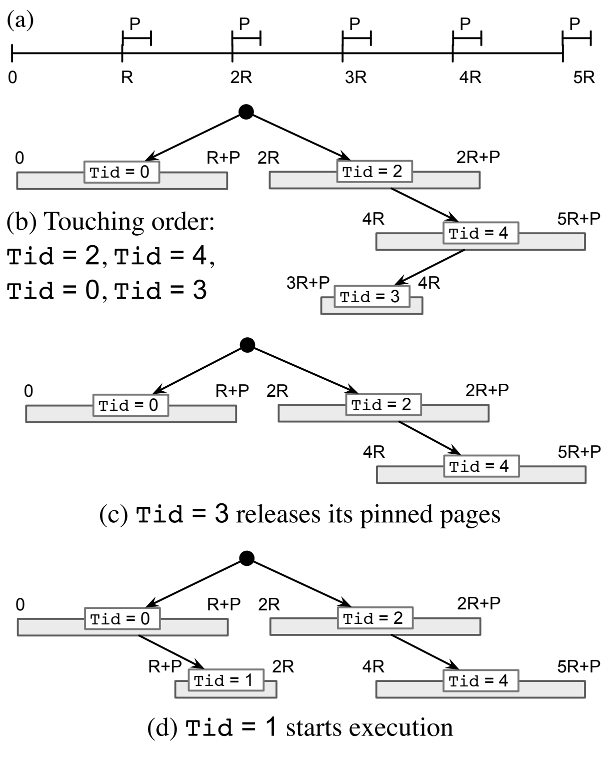

(a)
P
0
R
2R
3R
4R
5R
R+P
2R+P
4R
5R+P
3R+P
Tid = 0
Tid = 2
Tid = 4
Tid = 3
Tid = 1
(b) Touching order: Tid = 2, Tid = 4, Tid = 0, Tid = 3
(c) Tid = 3 releases its pinned pages
(d) Tid = 1 starts execution

Figure 11: (a) The memory layout and data distribution after an initial round of execution of the program in Figure 3 (a). (b-d) Interval tree showing how pinned pages are organized at different moments of the program execution.

Figure 11 shows how we organize the pinning of pages during a round of execution of our running example, seen in

Figure 3 (a). In this example, threads of identifier `Tid` and `Tid + 1` access overlapping memory regions through $A[i]$ in lines 2 and 10 of Figure 3 (a). Thus, it is possible that one thread will try to migrate a memory page that has been copied by another thread. To avoid this situation, before performing this copy, a thread t must search for the range of data to be copied in our interval tree. If the tree does not contain this range, then t is allowed to perform the migration, and a new entry is added to the tree. On the other hand, if the tree contains part of the range, then a partial migration might take place. As an example, Figure 11 (d) shows the configuration of the interval tree once thread `Tid = 1` attempts to copy pages over. Part of the requested range has been pinned by threads `Tid = 0` and `Tid = 2`. Therefore, thread `1` will be allowed to migrate a range of pages that is smaller than its original request.

4. EXPERIMENTAL RESULTS

The primary goal of this section is to demonstrate that our technique is able to speedup code without human intervention. Most of our techniques rely on our ability to predict the trip count of loops. Thus, a secondary goal of this section is to show that we can perform such predictions reliably. We start the section by diving into this second goal and leave the latter for Section 4.2. Given these two distinct goals, Sections 4.1 and 4.2 use two different benchmark suites. In Section 4.1 we measure the precision of our trip count predictor in the programs of the SPEC CPU 2006 benchmark suite. We use SPEC because this benchmark gives us an extensive number of loops, which we can analyze statically, and measure dynamically. In Section 4.2 we use a suite of parallel benchmarks, which is distributed with our extension of the LLVM compiler. Even though our technique is applicable to sequential code - for example reducing memory access latencies bringing data from remote memory closer to the core where it is needed - the parallel benchmarks give us more opportunities to exercise all the features of our technique.

4.1 Trip Count Prediction

Actual loops have simple topology. Table 1 shows structural characteristics of the loops that we found in SPEC CPU 2006. SPEC gave us a total of 21,336 loops, out of which we could instrument 11,850, i.e., 55.54%. We fail to instrument a loop when we do not find its induction variable. This failure might happen for three main reasons: (i) the loop is controlled by an oscillating induction variable; (ii) it is controlled by a function call; or (iii) it is controlled by data located in memory. We have not found examples of (i) in SPEC. Situation (ii) happens in loops like `while(f());` an example of (iii) is `for (p = a; *p != '0'; p++)`.

Table 1 also shows the number of Strongly Connected Components (SCC) formed by the different induction variables that we found in the loops. This number is usually larger than the number of loops, because some of them might be controlled by more than one induction variable. We have found a few linear monotonic variables that might be incremented through different paths during the execution of the loop. Nevertheless, this number is rather small: 16% of the induction variables present this characteristic.

Hit rate of the predictor. We have developed a profiler that measures the actual trip count of loops. The result of our profiler lets us observe how accurate our heuristic

Program	**L**	**IL**	**%IL/L**	**SCC**	**SP**
milc	426	391	91.78%	426	409
namd	623	469	75.28%	781	604
dealII	6,526	3,535	54.17%	7,249	6,077
soplex	742	422	56.87%	807	683
lbm	23	23	100.00%	24	23
bzip2	238	186	78.15%	244	171
gcc	4,614	1,798	38.97%	5,121	4,513
mcf	50	7	14.00%	54	40
gobmk	1,288	1,040	80.75%	1,555	1,283
hmmer	881	664	75.37%	946	825
sjeng	267	106	39.70%	276	221
libquantum	98	77	78.57%	123	100
h264ref	1,870	1,411	75.45%	1,946	1,841
omnetpp	465	238	51.18%	470	379
astar	119	80	67.23%	138	118
xalancbmk	3,106	1,403	45.17%	3,024	2,486
Total	21,336	11,850	55.54%	23,184	19,773

Table 1: L: number of loops. IL: number of loops that we could instrument. SCC: number of induction variables. SP: induction variables having a unique increment.

estimates of Section 3.3 are. We have split our accuracy results into seven categories according to the actual number N of iterations:

- $[0,\ \sqrt{N}]$: Occurs when the estimated trip count is less than or equal to the square root of the actual trip count. For example, if we estimate that a loop will iterate 2 times and it iterates 10 times during its execution, this loop will be classified into this category.
- $]\sqrt{N},\ N/2]$: Occurs when the estimated trip count lies between the square root of the actual trip count and half its value. This case happens, for instance, if we estimate that a loop will iterate 4 times, but it actually iterates 10 times.
- $]N/2,\ N[$: Occurs when the estimated trip count lies between halve the actual trip count and the trip count itself. For example, if we estimate that a loop will iterate 8 times and it iterates 10 times during the actual execution, this loop will be classified into this category.
- $[N,\ N]$: Occurs when the estimated trip count equals the actual trip count.
- $]N,\ 2*N]$: Occurs when the estimated trip count is greater than the actual trip count, but is less than or equal to twice its true value. For example, if we estimate that a loop will iterate 16 times, but it actually iterates 10 times, this instance of the loop execution will fall into this category.
- $]2*N,\ N^2]$: Occurs when the estimated trip count is greater than twice the actual trip count, but is less than or equal to the square of the actual trip count. This case happens, for instance, if we estimate that a loop will iterate 32 times and it iterates 10 times in fact.

Program	$[0, \sqrt{N}]$	$]\sqrt{N}, N/2]$	$]N/2, N[$	$[N, N]$	$]N, 2N]$	$]2N, N^2]$	$]N^2, +\infty]$
SPEC	15,914,068	1,113	2,877,992	955,866,120	44,786,968	6,026,735	37,227,397
Subtotal (%)	1.50%	0.00%	0.27%	89.95%	4.21%	0.57%	3.50%
LLVM	25,525,142	2,078	2,922,080	4,134,074,825	163,974,403	11,363,892	400,209,181
Total (%)	0.54%	0.00%	0.06%	87.25%	3.46%	0.24%	8.45%

Table 2: Precision of the trip count analysis of Section 3.3.

- $]N^2, +\infty]$: Occurs when the estimated trip count is greater than the square of the actual trip count. For example, if we estimate that a loop will iterate 128 times, but it actually iterates 10 times, this run will fall into this category.

Table 2 shows the comparison between the estimated trip count and the actual trip count collected via profiling. "SPEC" summarizes the results for the SPEC CPU benchmarks, while "LLVM" also includes more than 430 benchmarks distributed with the LLVM test suite. While running the programs, each time a loop stops, we collect the actual trip count and compare it with the estimated trip count. Thus, the numbers that we present gives us the quantity of *dynamic instances* of loops during runtime, instead of the number of natural loops in the code. We did this because we may predict correctly the trip count of some executions of a loop, but may predict wrongly the trip count of other instances of the same loop.

4.2 Performance Analysis

In this section we compare the relative performance of four different page allocation policies:

1. **Selective Page Migration**: the technique that we propose.

2. **Minas**: a middleware plus a library that has been developed for NUMA aware architectures [25, Ch.5]. Minas replaces the memory allocation calls of the C standard library with an API of its own [26]. Thus, it decides, at runtime, where memory pages will be placed, based on two policies: first-touch and round robin. Threads are pinned to the cores where they start running. Once a thread asks for memory, it is given space on its local NUMA node. If this node already contains a certain number of active memory pages, then the next requests are scattered around remote memory banks, in round-robin fashion, to decrease memory contention. Minas performs page placement whenever a thread asks for memory, but it does not migrate a page after its creation.

3. **Premature migration**: a policy in which pages are always migrated to the cores in which they are accessed. This migration happens regardless of reuse rate or array size.

4. **Baseline**: a policy that never migrates pages. Once allocated, a page remains in its original memory bank until either it is evicted due to usual page thrashing, or the program terminates. The page allocation policy is first-touch.

We run experiments in a 64-core ccNUMA computer featuring four 16-core AMD Opteron 6282 SE processors. This machine has 8 nodes/memory controllers, each one connected to a memory bank with 16GB of memory. Cores inside a processor are organized in eight pairs. The two cores inside a compute unit share a 64KB L1 instruction cache, a 2MB L2 cache, a floating-point unit (FPU), and a fetch/decode unit. Each core has its own 16KB L1 data cache and integer execution pipeline. The compute units share a 6MB L3 cache. All the performance experiments that we show in this section use all the 64 available cores. Our operating system is Ubuntu 12.04.1 LTS, kernel 3.2.0-23 x86_64.

In order to compare these different page allocation policies we apply them onto six different parallel benchmarks. To ensure reproducibility, these six benchmarks are shipped together with our distribution of the LLVM compiler and are freely available in our repository. The benchmarks were designed and implemented in a way that is oblivious to the memory allocation policy. All these programs implement typical parallel algorithms using C POSIX Threads. Four of our benchmarks are linear algebra applications: matrix multiplication, Cholesky decomposition, LU decomposition, and matrix addition. A fifth benchmark is an implementation of the K-Nearest Neighbors (KNN) data-mining algorithm. Our last benchmark is the parallel bucket sort algorithm. Figure 12 shows our runtime results for these programs.

We first notice that in the case of matrix addition, selective migration always beats premature migration. The reuse rate of each array access is small, since each element is only accessed once. Our heuristic never migrates any page, and thus, it is only slightly worse than the original implementation of the algorithm (baseline), due to its runtime overhead. Nevertheless, a quick inspection of Chart (e), in Figure 12, reveals that this overhead is barely noticeable, as we have been able to hoist the dynamic check to outside the outermost loop in the code. On the average, selective migration is 3% slower than the original algorithm. On the other hand, premature migration always sends pages over, paying the full price of a data transfer, to communicate values that will be read only once. Minas also always beats the premature migration policy; however, its overhead is considerably larger than our technique.

Premature and selective migration yield similar results when applied on LU decomposition, Cholesky decomposition and matrix multiplication, given large inputs. In this case, both approaches migrate the two arrays that are accessed more often. Nevertheless, our heuristic is better than the premature approach for small matrices. This happens because it does not pay off to migrate arrays that are too small, as they fit entirely in cache. Hence, the main memory is only accessed a few times. For very large matrices, our technique yields smaller speedups than premature mi-

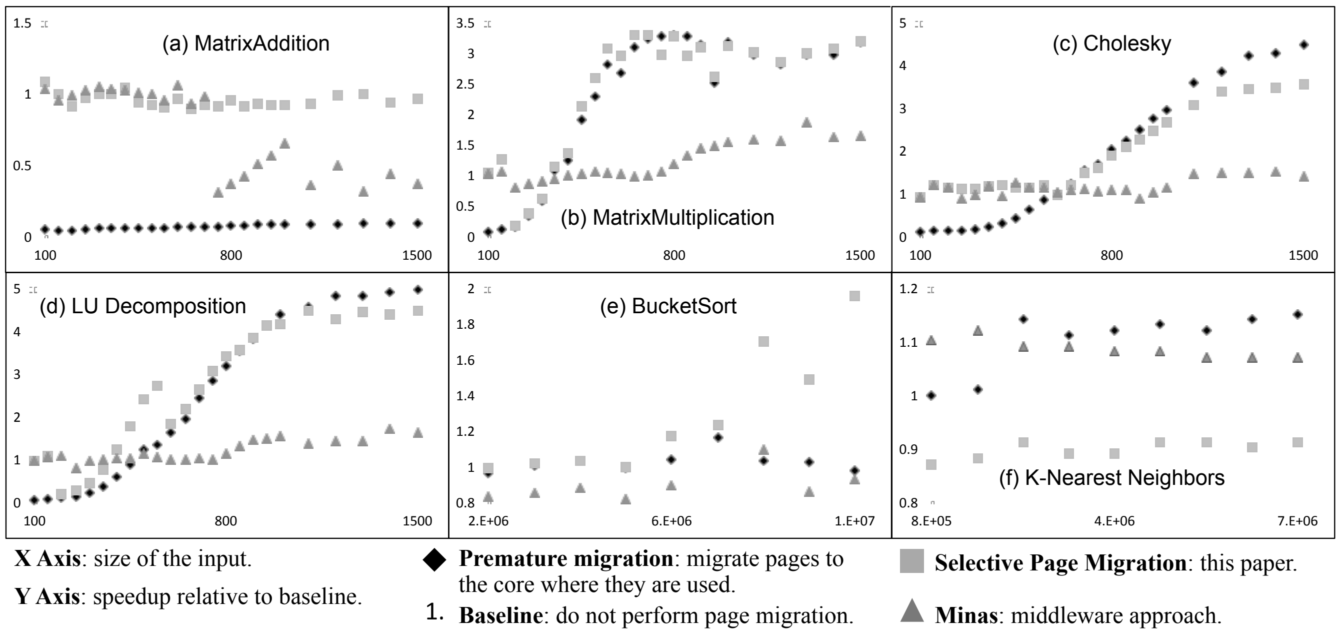


Figure 12: Relative performance between four different page allocation policies. Input size for: (a-d) side of square matrix; (e) size of array to be sorted; (f) size of array of 2D points that constitutes the search space of K-Means.

gration. We believe that this happens due to the overhead of keeping a large number of pages in the interval tree.

Parallel bucket sort gives us a promising result: our heuristic is always better than premature migration. This algorithm has four steps. First it sets up an array of initially empty buckets. Second, it scatters the elements to be sorted into buckets, in such a way that the elements of the first bucket are smaller than the elements of the second bucket, and so on. Third, it sorts the elements of each bucket in parallel. Finally, it merges the elements of the buckets to obtain the complete list of sorted elements. The premature migration policy attempts to migrate pages during all the four phases of the algorithm, whereas we can recognize the small reuse rate of array accesses in all the phases but the third. Thus, in this case we migrate data only when this operation is beneficial to our algorithm.

Finally, our last chart, seen in Figure 12 (f), shows a situation when our heuristic should migrate a page, but it does not - in this case, we deem the amount of array reuse in the K-Nearest Neighbors algorithm too small to justify the page transfer. However, premature migration reveals that this conservative behavior is a mistake in this scenario, given that it delivers gains of up to 15%. Our overhead causes, one the average, a 10% slowdown in the application. This overhead is larger than in the case of matrix addition, shown in Figure 12 (a), because we have not been able to hoist the dynamic test completely outside the outermost loop of KNN.

5. RELATED WORK

To the best of our knowledge, the analyses and optimizations that we describe in Section 3 are novel. However, we are aware of the seminal work of Wolf and Lam [30], which laid the foundations of much of the practice and science of loop analyses. In our work, we preferred to approach loop analysis using symbolic techniques, because we had to generate formulas describing trip counts and array sizes. It is not clear if such formulas could be produced using Wolf and Lam's methods. A work more similar to ours is due to Chatterjee *et al.* [9], which generates exact formulas to bound array regions. Our technique can be refined with more precise models; after all, our formulas give us only estimates, not exact numbers. However, we emphasize that our algorithms are very efficient. Whereas Chatterjee's relational analysis is exponential on the number of bounds, our method is quadratic.

There are techniques to optimize data placement in NUMA aware architectures. These methods speedup programs by reallocating memory pages and mapping threads to cores. Reallocation and mapping reduce load balancing issues, memory contention and remote accesses on NUMA machines [15, 6, 26]. In this section we group the most well known techniques into different categories, depending on which level of the runtime environment they are implemented: hardware, operating system, middleware and libraries.

Hardware. Tikir and Hollingsworth [28] have introduced a profile-driven mechanism based on hardware counters to monitor the memory access behavior of an application. The profile is used to decide whether memory pages should be migrated or not. Cruz *et al.* [10] have proposed a mechanism to detect the communication pattern between threads. Their technique exploits the Translation Lookaside Buffer to obtain the necessary information to map threads to cores. The resulting mapping reduces the communication latency by providing data locality to threads. The work of Dykema *et al.* [15] has introduced a mechanism that implements synchronization for data transfers between two machines with independent memory controllers. Dykema *et al.*'s goal is to use the communication link between the nodes to synchronize memory and to reduce the overhead of data transference. In contrast to these works, we focus on a general solution that does not rely on a specific feature of a target architecture. In other words, this paper proposes a way to speedup applications that does not require modifications in the current NUMA architectures to be effective.

Operating System. Awasthi *et al.* [2] have proposed modifications in the operating system memory allocator to take into account the NUMA design. Their modifications improve virtual-to-physical mappings for every page fault by reducing memory access delays. Migration is performed during runtime and on spare cycles. Löf and Holmgren [23] have introduced a new memory policy named affinity-on-next-touch for the Solaris operating system. This policy allows data migration when threads touch pages on the next time. Thus, the threads can have their data in the same node, allowing more local accesses. Blagodurov *et al.* [4] have introduced a contention-aware scheduling algorithm for NUMA systems. Their work shows how contention-aware algorithms designed for UMA architectures hurt the performance of NUMA multicore systems. Based on these findings, the authors propose an algorithm that minimizes both thread and data migration, while keeping data locality. The proposal has been implemented as a user-level scheduler for the Linux operating system. Diener *et al.* [12] have presented a mechanism to dynamically map threads to machine cores. This technique detects the communication pattern of the application by monitoring a table that logs page accesses. Using such information, their approach migrates threads in order to keep closer the threads that communicate directly. This solution has been implemented inside the Linux kernel. Closer to our work, Li *et al.* [22] have used compiler analyses to determine data placement. Memory allocation, in their case, uses a customized implementation of the `malloc` function, and requires system support: an extra field in memory pages is used to determine its owner. Contrary to these previous work, our solution does not depend on a specific operating system. Even though we have implemented it in Linux, with the `hwloc` library, it will work in any system that gives us: (i) information about the topology of the underlying architecture and (ii) a way to migrate pages.

Middleware. Broquedis *et al.* [6] have proposed a runtime system for OpenMP. This runtime system, called ForestGOMP, groups threads by affinity and performs data migration to reduce NUMA impact on OpenMP applications. Thread grouping and data migration decision is based on hints provided by the OpenMP directives of the application. The developer does not have to modify the application source code, however, the approach only works for OpenMP applications. Castro *et al.* [8] have proposed a machine learning based approach that finds an efficient thread mapping for NUMA systems. This approach takes into account both application and platform characteristics. It relies on software transactional memory to hide from the application developer all mapping decisions. Pilla *et al.* [24] have proposed a load balancing algorithm that migrates tasks over NUMA domains. It equalizes load and guarantees memory affinity by getting runtime information from the application. The algorithm is implemented inside the Charm++ parallel runtime system [19] and does not require code modification. Our solution differs from these previous works because it does not require a target runtime system. Additionally, our technique is more aware of the specific structure of the target program, as it is implemented at the compiler level. Even though we have not compared our approach against all these systems, we believe that our overhead is lower, because we perform most of our computations statically, leaving only fast runtime checks to be executed dynamically.

Library. Library solutions are closely related to the *middleware* category, which we have described previously, but, contrary to it, in this case programmers must change their code using particular APIs. Ribeiro *et al.* [26] presented an API and a runtime environment to allocate and place data over the NUMA machine. This API abstracts to the developer the topology of the architecture and offers mechanisms to determine the initial allocation and placement of application data. Wittmann and Hager [29] presented a software layer that reduces adverse effects of task distribution on ccNUMA systems by sorting tasks into locality queues. Each of these queues is preferably processed by threads that belong to the same locality domain, or node. Dupros *et al.* [14] optimized a parallel seismic wave propagation software for NUMA machines. This optimization was performed via direct modifications of the application to be aware of data allocation and placement. The overall performance of the application has been improved significantly due to the programmer's intervention. Borin and Devloo [5] have discussed the issues of implementing parallel finite elements methods on multicore NUMA machines. Both Dupros *et al.* [14] and Borin and Devloo [5] have modified their application to explicitly manage data and thread placement. Such management has been done via the `numactl` tool and the `libnuma` API. They also have achieved very expressive speedups. Contrary to these previous works, our solution does not require any modification in the source code of the application. Furthermore, our method is not program specific and can be applied to different ranges of applications.

6. CONCLUSION

We have presented a compiler-based technique to improve page placement in NUMA machines. Our approach joins static and dynamic analyses in an effective way to allow parallel benchmarks to benefit from non-uniform memory architectures. We have shown that it is possible to speed software's performance up to four times without programmer's intervention. We are not aware of other solutions that are completely restricted to the compiler: we did not have to modify the operating system nor the memory allocation API used in the C standard library.

Limitations and Future Work. Our implementation currently does not handle OpenMP directives, as they have a particular semantics that we have not yet encoded in our framework. At this time, we only analyze the syntax and semantics that is standard in the intermediate representation of LLVM, our baseline compiler infrastructure. Given the vast amount of benchmarks written in OpenMP, we intend to incorporate its semantics in our analyses.

Software. All the techniques that we have implemented in this paper, plus the benchmarks that we have used in Section 4.2 are publicly available at `http://code.google.com/p/selective-page-migration-ccnuma`.

7. ACKNOWLEDGMENTS

We thank the PACT referees for their very constructive comments. The Unicamp researchers are supported by CAPES, CNPq, FAPESP and Petrobras. The UFMG researchers are supported by FAPEMIG, CAPES and CNPq. Raphael was partially supported by Intel. We thank the LabMeC (Laboratório de Mecânica Computacional) team for providing us with the machine used in the experiments.

8. REFERENCES

[1] A. W. Appel and J. Palsberg, *Modern Compiler Implementation in Java*, 2nd ed. Cambridge University Press, 2002.

[2] M. Awasthi, D. W. Nellans, K. Sudan, R. Balasubramonian, and A. Davis, "Handling the problems and opportunities posed by multiple on-chip memory controllers," in *PACT*. ACM, 2010, pp. 319–330.

[3] S. Basu, R. Pollack, and M.-F. Roy, *Algorithms in Real Algebraic Geometry*. Springer, 2006.

[4] S. Blagodurov, S. Zhuravlev, A. Fedorova, and A. Kamali, "A case for NUMA-aware contention management on multicore systems," in *PACT*. ACM, 2010, pp. 557–558.

[5] E. Borin and P. Devloo, "Programming finite element methods for ccNUMA processors," in *Int. Conference on Parallel, Distributed, Grid and Cloud Computing for Engineering*. Civil-Comp Press, 2013.

[6] F. Broquedis, N. Furmento, B. Goglin, P.-A. Wacrenier, and R. Namyst, "ForestGOMP: An efficient OpenMP environment for NUMA architectures," *Inter. J. Parallel Programming*, vol. 38, no. 5-6, pp. 418–439, 2010.

[7] F. Broquedis, J. Clet-Ortega, S. Moreaud, N. Furmento, B. Goglin, G. Mercier, S. Thibault, and R. Namyst, "hwloc: A generic framework for managing hardware affinities in HPC applications," in *PDP*. IEEE, 2010, pp. 180–186.

[8] M. Castro, L. F. W. Góes, C. P. Ribeiro, M. Cole, M. Cintra, and J.-F. Méhaut, "A machine learning-based approach for thread mapping on transactional memory applications," in *High Performance Computing Conference (HiPC)*. Bangalore, India: IEEE, 2011, pp. 1–10.

[9] S. Chatterjee, E. Parker, P. J. Hanlon, and A. R. Lebeck, "Exact analysis of the cache behavior of nested loops," in *PLDI*. ACM, 2001, pp. 286–297.

[10] E. Cruz, M. Diener, and P. Navaux, "Using the translation lookaside buffer to map threads in parallel applications based on shared memory," in *IPDPS*, 2012, pp. 532–543.

[11] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck, "Efficiently computing static single assignment form and the control dependence graph," *TOPLAS*, vol. 13, no. 4, pp. 451–490, 1991.

[12] M. Diener, E. Cruz, and P. Navaux, "Communication-based mapping using shared pages," in *IPDPS*, 2013, pp. 700–711.

[13] E. W. Dijkstra, "A note on two problems in connexion with graphs," *Numerische Mathematik*, vol. 1, pp. 269–271, 1959.

[14] F. Dupros, C. P. Ribeiro, A. Carissimi, and J.-F. Méhaut, "Parallel simulations of seismic wave propagation on NUMA architectures," in *PARCO*. IOS, 2009, pp. 67–74.

[15] G. L. Dykema, D. H. Bassett, and J. L. Lach, "Mechanisms for synchronizing data transfers between non-uniform memory architecture computers," 2012, US Patent 8,244,930.

[16] J. Ferrante, J. Ottenstein, and D. Warren, "The program dependence graph and its use in optimization," *TOPLAS*, vol. 9, no. 3, pp. 319–349, 1987.

[17] B. Goglin and N. Furmento, "Enabling high-performance memory-migration in Linux for multithreaded applications," in *MTAAP*. IEEE, 2009.

[18] D. Joyner, O. Čertík, A. Meurer, and B. E. Granger, "Open source computer algebra systems: SymPy," *ACM Commun. Comput. Algebra*, vol. 45, no. 3/4, pp. 225–234, 2012.

[19] L. V. Kale and A. Bhatele, Eds., *Parallel Science and Engineering Applications: The Charm++ Approach*. Taylor & Francis Group, CRC Press, Nov. 2013.

[20] C. Lameter, "An overview of non-uniform memory access," *Commun. ACM*, vol. 56, no. 9, pp. 59–54, 2013.

[21] C. Lattner and V. S. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *CGO*. IEEE, 2004, pp. 75–88.

[22] Y. Li, R. Melhem, A. Abousamra, and A. Jones, "Compiler-assisted data distribution for chip multiprocessors," in *PACT*. ACM, 2010, pp. 501–512.

[23] H. Löf and S. Holmgren, "affinity-on-next-touch: increasing the performance of an industrial PDE solver on a cc-NUMA system," in *ICS*. ACM, 2005, pp. 387–392.

[24] L. L. Pilla, C. P. Ribeiro, P. Coucheney, F. Broquedis, B. Gaujal, P. O. Navaux, and J.-F. Méhaut, "A topology-aware load balancing algorithm for clustered hierarchical multi-core machines," *Future Generation Computer Systems*, vol. 30, no. 0, pp. 191 – 201, 2014.

[25] C. P. Ribeiro, "Contributions on memory affinity management for hierarchical shared memory multi-core platforms," Ph.D. dissertation, University of Grenoble, 2011.

[26] C. P. Ribeiro, J.-F. Méhaut, and A. Carissimi, "Memory affinity management for numerical scientific applications over multi-core multiprocessors with hierarchical memory," in *IPDPS Workshops*. IEEE, 2010, pp. 1–4.

[27] L. Tang, J. Mars, X. Zhang, R. Hagmann, R. Hundt, and E. Tune, "Optimizing Google's warehouse scale computers: The NUMA experience," in *HPCA*. IEEE, 2013.

[28] M. M. Tikir and J. K. Hollingsworth, "Using hardware counters to automatically improve memory performance," in *Supercomputing*. IEEE, 2004, pp. 46–46.

[29] M. Wittmann and G. Hager, "Optimizing ccNUMA locality for task-parallel execution under OpenMP and TBB on multicore-based systems," 2011.

[30] M. E. Wolf and M. S. Lam, "A data locality optimizing algorithm," in *PLDI*. ACM, 1991, pp. 30–44.

[31] M. Wolfe, *High Performance Compilers for Parallel Computing*, 1st ed. Adison-Wesley, 1996.