On Instruction Organization

Tyler Dwyer
Simon Fraser University
tdwyer@sfu.ca

Alexandra Fedorova
Simon Fraser University
fedorova@sfu.ca

Abstract

To attain high program performance, a developer must be conscious to the many intricacies of hardware, and organize their code accordingly. This however, is not an easy task. Often the hardware is unknown to developers, or, if it is known, it is difficult to control or account for. Developers struggle with this challenge by using hardware conscious algorithms, specialized programming languages, or doing manual low-level optimizations.

We investigate the concept of instruction organization at a more general level. In particular, we investigate if a program, running on existing hardware, can be automatically reorganized according to a chosen organization metric. Further, if the reorganization can be done automatically, a program can then be reorganized during execution to adapt to changes in system resources, and fluctuating execution and data access patterns.

We use data locality as an organization metric with the goal of reducing data access latency and improving program performance.

1 Introduction

In the vast majority of cases, how a developer organizes his or her code into functions and threads **is not** how it should be executed. And, as this level of organization is beyond the scope of compiler optimizations, the organization structure here determines the organization, and thus execution, of the compiled program. An optimal organization would take both hardware specifications and execution characteristics into account, but this information is often difficult to incorporate into the program or is unavailable to the developer. Sub-optimal organization can lead to sub-optimal hardware utilization which has two serious side effects: performance degradation and wasted power.

One method to increase hardware utilization on current hardware is to increase data locality. Data locality is the solution to the problem of moving data to, from, and across a hardware context, such as a CPU. In short, data stored locally can be accessed faster and with less power than data stored remotely – often a difference of over an order of magnitude. This difference has led to the realization that both program performance and the future of hardware scaling hinges on reducing the quantity of data moved across and off chip [3, 7, 15]. For this reason, we use data locality as our example organization metric.

Achieving data locality however, can be a challenge. For example, Figure 1 demonstrates the common event of two threads updating a shared variable. Upon an update to the shared variable, each thread will check for a mutex, lock the mutex (if available), move the shared variable into its local memory, update it, and release the lock. This requires both the mutex data and the shared variable data to be transferred between cores. A better implementation of this example is to reorganize the instructions so that the shared variable is kept local and is updated at the request of the other thread – the reorganization step is shown in Figure 1c, with the result shown in Figure 1d. This has three benefits. First, the costly data migration has been swapped with a small, lightweight control migration. Second, the need for a mutex has been removed as only one thread accesses the shared data. And third, since the data never moves, maximal data locality is ensured, as indicated in Figure 1d. Here, both program organizations are functionally equivalent, but differ in how their instructions are organized within and across threads – a difference that can have a significant impact on performance. Lozi [11] and Soares [16] independently explored this principle in a database, and operating system, but in both cases the developer had to put significant effort into redesigning the system.

The previous example highlights how instruction organization can improve data locality, but this is just one metric of many that can be used to guide instruction organization. Another possible metric is instruction locality (I-cache reuse), shown in Figure 2. In this example, as is

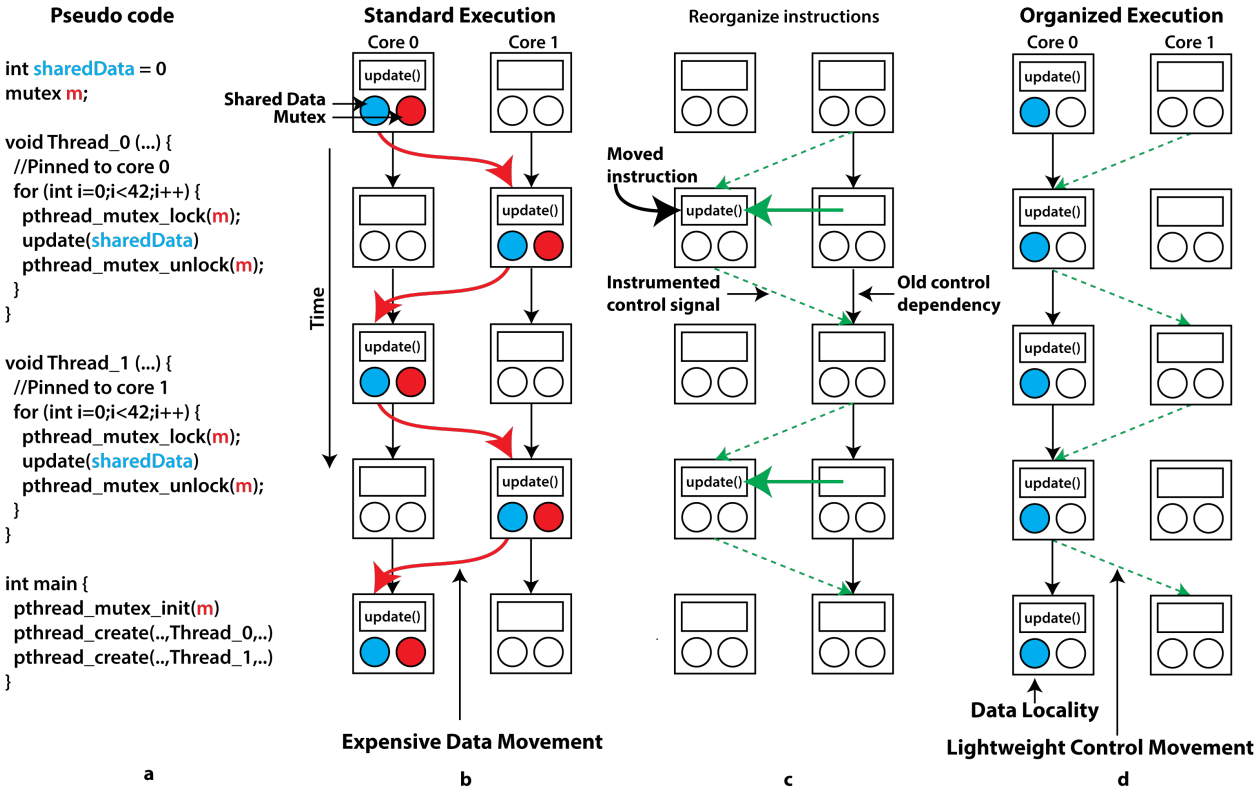

Figure 1: An example of how control migration can replace data migration to facilitate data locality. (a) shows the pseudo-code for a shared data object passed between two threads. (b) illustrates how this data, protected by a mutex, bounces back and forth between threads. (c) shows one way to reorganize the instructions; the update function of one thread is moved into the other thread and control signals are instrumented to execute the update remotely. This reorganization, as seen in (d), achieves maximal data locality, and removes the need for a mutex.

the case in many programs, a thread re-uses some code, but by the time it is re-executed, the required instructions have been evicted from the I-cache – Figure 2a. A better organization would split the code into pieces and spread the execution across many cores to take advantage of all available caches, shown in Figure 2b. Tözün et al. [17] took a similar approach of using control migration, but the solution was specific to databases.

A third metric in-line with technological trends, is to take advantage of single ISA heterogeneous systems. For these systems it is important to offload work to smaller, slower cores, but what work to offload and when to offload it is an area of active research [13]. This is an instruction organization problem for which an organization metric can be created, and our framework used to facilitate the intelligent management.

Unsurprisingly, implementing this reorganization is difficult. Related work has often addressed this issue in a problem-specific manner, by using new programming languages such as Charm++ [1], or developing new hardware such as TRIPS [14]. We believe that instruction organization, along with a means to execute the reorganized instructions, is the general concept behind all of this work.

The question we address is: can an existing program, running on existing hardware, be arbitrarily reorganized according to some metric (e.g., data locality), without assistance from the developer? Even more interestingly, if this reorganization can be done automatically, can a program be reorganized throughout its execution to adapt to changing execution and data access patterns? We believe the answer to both these questions is yes, and propose a method to do so.

We present a brief overview of our framework and challenges in Section 2 with discussion and relevant related work in Section 3.

2 Method

Our proposed method to organize instructions is best divided into three steps: getting the program's instructions

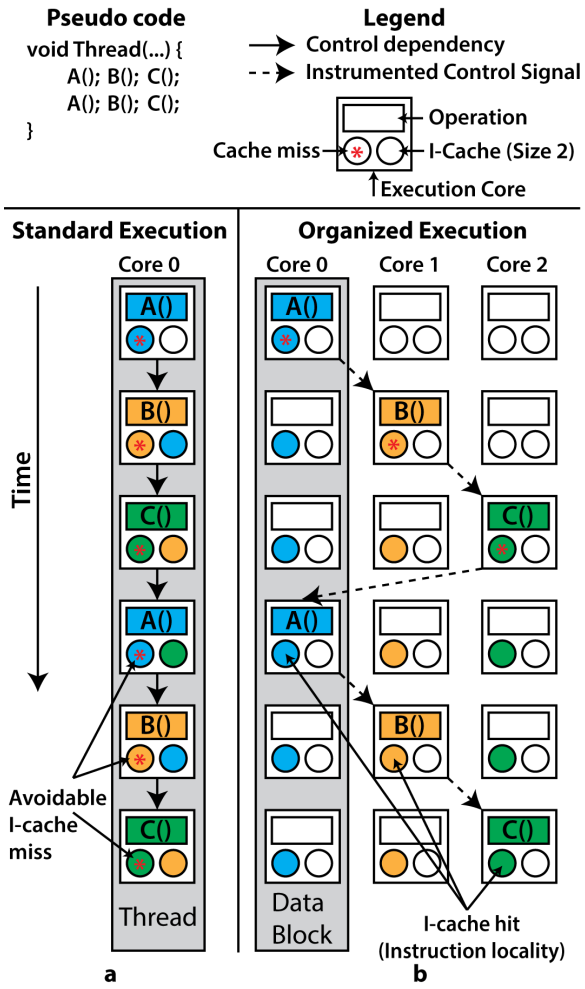

Figure 2: An example of how instruction organization can improve instruction locality. (a) shows a situation where three functions executing on one core cause continual instruction cache misses. (b) shows a better organization where functions are spread across cores to make use of multiple caches - instruction locality is achieved.

and determining their dependencies, organizing those instructions, and instrumenting a framework to execute the reorganized instructions.

2.1 Instructions & Dependencies

We use the LLVM framework [10] and perform our reorganization on LLVM's Intermediate Representation (IR) instructions; a simple example is shown in Figure 3ab. This IR is similar to a hardware independent type of machine instructions with an infinite register set – a program representation with both programming language and hardware specifics abstracted away.

Once a program is in IR form, we conservatively add all possible dependencies for all instructions across all functions and threads, creating a Program Dependency Graph (PDG), Figure 3c. Adding all possible dependencies will likely create many unnecessary dependencies, but it guarantees execution correctness and doesn't necessarily harm performance. The only detriment from these extra dependencies is that the instruction organization may not be optimal, which may lead to sub-optimal performance. The influence of these extra dependencies can be minimized through the use of optimizations, profiling information, and runtime information.

2.2 Organization

To organize instructions we create organization blocks, which we call *org blocks*, shown in Figure 3d. The size of these blocks is determined at compile time and is based upon: the block size to overhead trade off, instruction dependencies, instruction statistics such as the number of bytes allocated per block, and hardware information such as the LLC size.

In the running example of this paper we are reorganizing instructions to optimize for locality. Therefore, the composition of the blocks here is made to maximize internal instruction dependencies and minimize external instruction dependencies. This promotes instruction and data locality and reduces costly external communication. This is also a min-cut graph partitioning problem – how can the PDG be split into X pieces that minimize the number of dependencies cut through. Statically it is unknown which dependencies will be used more than others, so all are treated equally but more precise information can be added through profiling or further static analysis. Dependency edges inside an org block are handled by compiling/serializing the block. Similar to a thread, an org block can have multiple entries, multiple exits, and be instantiated multiple times during execution.

Once the org blocks are determined, the problem shifts from a static, compile time problem, to a dynamic, runtime problem. Organizing instructions dynamically implies that the instructions, or in our case org blocks, must be able to change where they are executed during the lifetime of the program - this is known as migration. When any grouping of instructions is migrated from one hardware context to another, it is a migration of control. Data migration often occurs along side this as the instructions that are migrated may rely on some data located on the originating core. Data migrations are generally very costly as all data must be moved, where as control migrations are often much cheaper as there are typically few instructions that need to be moved.

To implement dynamic organization we let every block decide, prior to its execution, whether it should be executed locally (from where it was called), or mi-

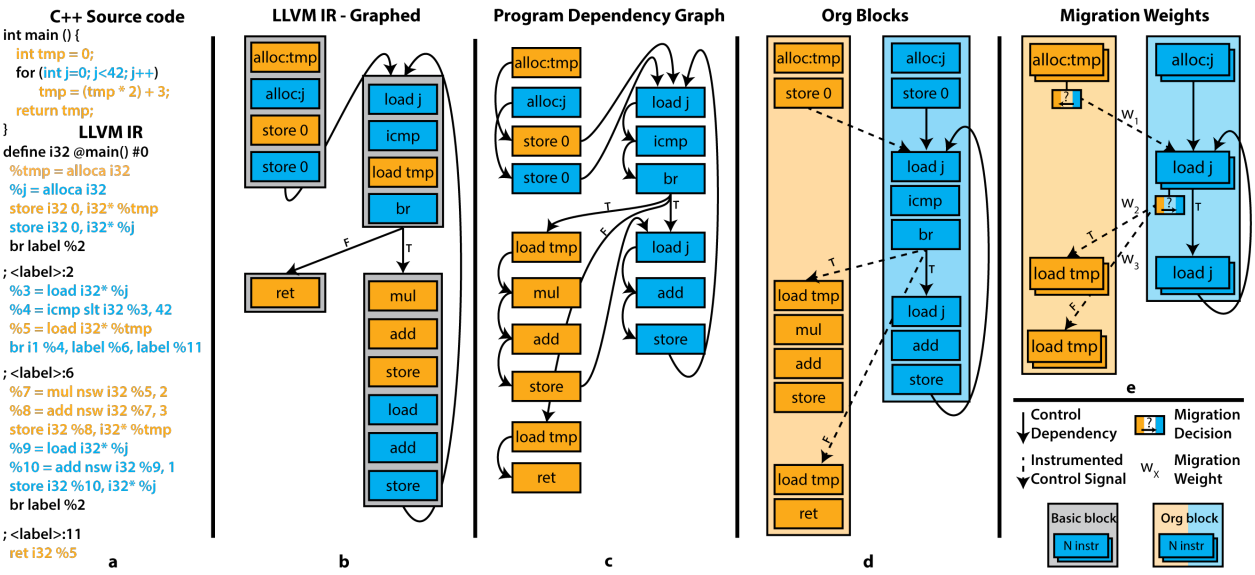

Figure 3: This outlines the proposed method to: convert C++ source code into an LLVM IR(a,b), find all necessary program dependencies (c), organize the instructions into organization (org) blocks and instrument control signals between blocks (d), and instrument the program with a framework to facilitate migration and execution (e).

grated to a different core. This decision is a comparison between two dynamically attained values. First is the current system state. For our running example of maximizing data locality, LLC miss rate is a good metric to use. Second is the program state, which is information about how the program is currently executing. We encode program state by adding a weight to all dependency edges *between* org blocks, shown in Figure 3e. When a dependency edge is used and spans two cores, say for a remote data access (data dependency edge), the weight for that edge is increased. When a dependency edge is used locally, say for a control signal between two local org blocks (control dependency edge), the weight may be decreased. The migration decision is then a comparison between the system state and summation of weights for that block. If the weights are low compared to system state (acceptably low remote communication), the block is not migrated and executed locally. If the weights are high compared to system state (high remote communication), the block is migrated.

The migration decision, weights, and weight update mechanism are all instrumented directly into each data block.

2.3 Framework instrumentation

The program is now organized, but in its current form it cannot be executed. The main reason for this is that there can exist multiple control dependencies from a single instruction; for example in Figure 3c, the 'br' instruction points to both the 'load tmp' and 'load j' instruction. Current hardware implementations are incapable of executing two instructions in parallel at this granularity. For this reason we must add an execution framework to facilitate these control dependencies. This framework is similar to the OS scheduler in that it starts the execution of code, but, as we are operating near the instruction granularity, any OS support is far too heavy for our needs. Instead, we instrument a lightweight scheduling, migration, and execution framework directly into the program. This framework manages control dependencies by signalling other org blocks to execute, and honours data dependencies by ensuring correct execution order.

2.4 Challenges

The two main challenges of this work are the graph cut algorithm and overhead.

The graph cut algorithm is responsible for automatically dividing the program into blocks and is dependent on the metric we are organizing for. The metric of data locality can be represented by reducing the number of cut dependencies between blocks – a min-cut problem. Min-cut graph problems are NP-complete, but as we do not require optimal cuts, heuristics can be used. Furthermore, as this step is done at compile time, a lengthy computation is not a significant concern. Other metrics will require different graph division algorithms and be based on possibly different information.

Our framework will incur overhead in three areas: the

overall overhead of the execution framework, overhead from control signals, and overhead from migration decisions and weight updates. This overhead however is only incurred during the already costly case of remote data or control migration. The common case of local execution is largely unaffected by our framework. Furthermore, as dynamic migration of blocks actively attempts to reduce remote migrations, it is also a mechanism to actively reduce all overhead incurred by the framework.

3 Discussion & Related Work

Our work touches on three disciplines of work: instruction organization, control migration, and runtime information.

There exist two ways to organize and execute instructions: by control and by data. Von Neumann architectures, such as x86/x64, are control based and organize instructions into blocks of varying granularity - basic blocks, functions, threads, etc. The execution of blocks is controlled by branch statements, and data dependencies are specified through programming language syntax (i.e. line order). Dataflow architectures, such as TRIPS [14], organize the execution of instructions according to data availability, and control can be specified through programming language syntax (i.e., control tokens) [6, 8].

Even though each organization method is radically different, they are functionally equivalent: control-flow programs can be converted into functionally-equivalent dataflow programs, and vice versa [2]. Our framework exploits this insight by taking a control-flow based application (written in C/C++), converting it to a dataflow form (the *Program Dependence Graph*, or PDG), organizing it with respect to some metric (e.g., data locality), and then reconverting it back into control-flow form for execution on current hardware. While we are not aware of any past work that takes this approach of instruction reorganization, Olden [4] and Charm++ [1] are examples of control flow languages that have a similar focus on organizing instructions according to data dependencies.

To facilitate a new instruction organization, we rely on scheduling and migration of small instruction blocks. Scheduling can be defined generally as managing when, and where to *start* execution of a block, whereas migration is the process of *moving* the control of a block from one core to another.

The effectiveness of any scheduler is largely determined by the granularity of tasks it is scheduling and the overhead incurred to schedule those tasks. Finer task granularity allows for better control over the system but yields higher overhead. Some schedulers have been designed to handle fine grained tasks at the millisecond scale [12], but nanosecond/cycle scale tasks would again incur prohibitive overhead. Our work allows for instruction-sized (very fine) task granularity and we instrument the scheduling framework directly into the program to avoid the heavy overhead of an OS scheduler.

Along with scheduling, we also instrument control migration to allow for dynamic reorganization of instructions during execution. Past work has used control migration as a tool to address problems such as improving D-cache utilization [3, 9], improving I-cache utilization [17], reducing lock contention [11], reducing system wide resource contention [18] and even minimizing system call overhead [16].

We use dynamic information to update migration weights which are used to drive the migration policy. Adding dynamic information, such as dynamic instruction statistics [5] or hardware monitors [18], into scheduling and migration policies allows for a application to adapt to both changing execution patterns and changes in system resources.

4 Conclusion

Currently it is the developers responsibility to organize their code for efficient execution – often an impossible task due to the lack of required information. Our method of instruction organization aims to relieve this responsibility, and perform it automatically based on much more information than the developer generally has access to. Our organization method uses static information gained at compile time such as dependencies and system specifications, runtime system information such as cache contention, and dynamic execution information such as access and execution patterns. This information determines how the program is subdivided into groups, where those groups start their execution, and when they migrate.

Our method divides a program into groups called org blocks. These blocks are created using an organization metric and dependency information to minimize inter-block communication and maximize intra-block locality – optimizing for data locality. The blocks are then instrumented with the ability learn about the dynamic system through weight updates. Finally a framework is instrumented into the program to facilitate the scheduling, migration, and execution of the org blocks.

We believe this technique can be used to reorganize an application to allow for efficient execution on any hardware for which an organization metric can be constructed. Our work focuses on data locality due to its relevance to current systems, but other metrics can be used.

References

[1] Bilge Acun, Abhishek Gupta, Nikhil Jain, Akhil Langer, Harshitha Menon, Eric Mikida, Xiang Ni, Michael Robson, Yanhua Sun, Ehsan Totoni, and Others. Parallel Programming with Migratable Objects: Charm++ in Practice. *Supercomputing*, 2014.

[2] Micah Beck, Richard Johnson, and Keshav Pingali. From control flow to dataflow. *JPDC: Journal of Parallel and Distributed Computing*, 1991.

[3] Silas Boyd-Wickizer, Robert Morris, M Frans Kaashoek, et al. Reinventing scheduling for multicore systems. In *HotOS*, 2009.

[4] Martin C. Carlisle and Anne Rogers. Software caching and computation migration in Olden. *PPoPP: Principles and practice of parallel programming*, 1995.

[5] Koushik Chakraborty, Philip M. Wells, and Gurindar S. Sohi. Computation spreading: employing hardware migration to specialize CMP cores on-the-fly. *SIGOPS: Operating Systems Review*, 2006.

[6] Jack Dennis and David Misunas. A preliminary architecture for a basic data-flow processor. *ACM SIGARCH Computer Architecture News*, 1975.

[7] Hadi Esmaeilzadeh, Emily Blem, Renee St. Amant, Karthikeyan Sankaralingam, and Doug Burger. Dark silicon and the end of multicore scaling. *ISCA: International Symposium on Computer Architecture*, 2011.

[8] Wesley M. Johnston, J. R. Paul Hanna, and Richard J. Millar. Advances in dataflow programming languages. *ACM Computing Surveys*, March 2004.

[9] Md Kamruzzaman, Steven Swanson, and Dean M. Tullsen. Software data spreading: leveraging distributed caches to improve single thread performance. *PLDI: Programming Languages Design and Implementation*, 2010.

[10] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. *CGO: International Symposium on Code Generation and Optimization*, 2004.

[11] Jean-Pierre Lozi, Florian David, Gaël Thomas, Julia Lawall, and Gilles Muller. Remote Core Locking: Migrating Critical-Section Execution to Improve the Performance of Multithreaded Applications. *USENIX ATC: Annual Technical Conference*, 2012.

[12] Kay Ousterhout, Patrick Wendell, Matei Zaharia, and Ion Stoica. Sparrow: Scalable scheduling for sub-second parallel jobs. *EECS Department, University of California, Berkeley*, 2013.

[13] Juan Carlos Saez, Manuel Prieto, Alexandra Fedorova, and Sergey Blagodurov. A comprehensive scheduler for asymmetric multicore systems. *EuroSys: European conference on Computer systems*, 2010.

[14] Karthikeyan Sankaralingam, Ramadass Nagarajan, Haiming Liu, Changkyu Kim, Jaehyuk Huh, Doug Burger, Stephen W Keckler, and Charles R Moore. Exploiting ilp, tlp, and dlp with the polymorphous trips architecture. In *Computer Architecture, 2003. Proceedings. 30th Annual International Symposium on*, pages 422–433. IEEE, 2003.

[15] John Shalf, Sudip Dosanjh, and John Morrison. Exascale computing technology challenges. In *VECPAR: High Performance Computing for Computational Science*, 2010.

[16] Livio Soares and Michael Stumm. FlexSC: Flexible System Call Scheduling with Exception-Less System Calls. *OSDI: Symposium on Operating Systems Design and Implementation*, 2010.

[17] Pinar Tözün, Islam Atta, Anastasia Ailamaki, and Andreas Moshovos. ADDICT : Advanced Instruction Chasing for Transactions. *VLDB: International Conference on Very Large Databases*, 2014.

[18] Sergey Zhuravlev, Sergey Blagodurov, and Alexandra Fedorova. Addressing shared resource contention in multicore processors via scheduling. *ASPLOS: International Conference Architectural Support for Programming Languages and Operating Systems*, 2010.