Article

A LLVM JIT Prototype for Running an Energy-Saving Hardware-Aware Mapping Algorithm on C/C++ Applications that use Pthreads

Iulia Știrb [*] and Gilbert-Rainer Gillich

Posted Date: 26 June 2023

doi: 10.20944/preprints202306.1834.v1

Keywords: LLVM; JIT compiler; number of cores; number of logical cores; NUMA systems; thread classification algorithm; thread mapping algorithm

Article

A LLVM JIT prototype for running an energy-saving hardware-aware mapping algorithm on C/C++ applications that use Pthreads

Iulia Știrb [1,*][0000-0002-9579-2201], **Gilbert-Rainer Gillich** [1,2][0000-0003-4962-2567]

[1] Department of Engineering Science, Babes-Bolyai University, P-ta Traian Vuia 1-4, 320085 Resita, Romania; iulia.stirb@gmail.com (I.S.)
[2] Doctoral School of Engineering, Babes-Bolyai University, P-ta Traian Vuia 1-4, 320085 Resita, Romania; gilbert.gillich@ubbcluj.ro (G.R.G.)
* Correspondence: iulia.stirb@gmail.com

Abstract: Low-Level Virtual Machine (LLVM) compiler infrastructure is a useful tool for building Just-in-time (JIT) compilers, besides its reliable front-end represented by clang compiler and its elaborated middle-end containing different optimizations that improve the runtime performance. This paper addresses specifically the part of building a JIT compiler using LLVM with the scope of getting the hardware architecture details of the underlying machine such as the number of cores and the number of logical cores per processing unit and providing them to NUMA-BTLP static thread classification algorithm and to NUMA-BTDM static thread mapping algorithm. Afterwards, the hardware-aware algorithms are run by the JIT compiler within an optimization pass. JIT compiler in this paper is designed to run on a parallel C/C++ application (which creates threads using Pthreads), before the first time the application is executed on a machine. To do that, the JIT compiler takes the native code of the application, gets the corresponding LLVM IR (Intermediate Representation) for the native code and executes the hardware-aware thread classification and the thread mapping algorithms on the IR. The NUMA-Balanced Task and Loop Parallelism (NUMA-BTLP) and NUMA-Balanced Thread and Data Mapping (NUMA-BTDM) are expected to optimize the energy consumption up to 15%, on NUMA systems.

Keywords: *LLVM, JIT compiler, number of cores, number of logical cores, NUMA systems, thread classification algorithm, thread mapping algorithm*

1. Introduction

Through LLVM compiling infrastructure [1], one can build JIT compilers which are used to interpret LLVM IR (Intermediate Representation) corresponding to language constructions that can be invented by the author. LLVM JIT [1] can also trigger LLVM optimizations on the LLVM IR constructs. JIT compiler prototype in this paper does not interpret new language constructs, but it is rather used to trigger an optimization pass which executes NUMA-BTDM [2] and NUMA-BTLP [3] algorithms on the LLVM IR representation of the code. The algorithms get the hardware details of the underlying architecture (number of cores and number of logical cores per processing unit) and perform a hardware-aware thread mapping.

NUMA-BTLP [3] is a static thread classification algorithm implemented using LLVM [1], which defines three types of threads depending on static criteria, namely autonomous threads, side-by-side threads and postponed threads [2]. The algorithm associates a thread with a tree node and adds the thread in a communication tree which defines the way threads communicate with each other by their position in the tree e.g. if thread *i* has data dependencies with thread *j*, than thread *i* could be the parent of thread *j* in the communication tree. Once having constructed the communication tree, the tree is traversed and each thread is mapped depending on its type which is stored in the corresponding node in the communication tree [4]. The thread is mapped to one or several processing units, based on the mapping rules defined by NUMA-BTDM algorithm [2] implemented using LLVM [1] e.g. autonomous threads, that have no data dependencies with any other thread, are distributed uniformly to processing units, side-by-side threads are mapped on the same processing units as other threads with which the share data, while postponed threads are mapped on the less loaded processing unit so far during the mapping process (the processing unit is determined statically) [4].

For the NUMA-BTLP [3] and NUMA-BTDM [2] optimization algorithms to have any effect, the parallel execution model used by the application is supposed to be Pthreads [5]. That is because the NUMA-BTLP algorithm [3] searches for `pthread_create` calls [5], which create the threads, in the LLVM IR [1] representation of the input code, to be able to get the function executed by the thread and perform the static data dependency analysis on it, resulting the thread type [4].

The issue is that, having these algorithms implemented in LLVM [1] involves getting the hardware details at compile-time and having the application optimized for the same machine where it was compiled. Therefore, the optimization would not have any effect on other machines than the one for which the application was compiled. A solution to this is constructing a JIT with LLVM [1] and triggering the thread classification and thread mapping algorithms by the JIT, as an optimization pass, anytime the application is first run on a machine.

This research is a case study of building a LLVM JIT for running NUMA-BTLP [3] and NUMA-BTDM [2] algorithms on the LLVM IR [1] of C/C++ applications that use Pthreads parallel model for the scope of saving energy by applying a hardware-aware thread mapping.

In previous researches [2-4], it is shown how the NUMA-BTLP [3] and NUMA-BTDM [2] algorithms improve the energy consumption by inserting a Pthreads thread mapping call after each Pthreads thread creation call, that maps the thread by taking into account the number of cores and the number of logical cores per processing unit, in establishing the affinity of the thread. The affinity of a thread is defined as the cores on which the thread runs, in Pthreads parallel model.

The advantages of using the hardware-aware mapping in [4] is that the threads that share data can be are mapped on the same cores and the threads that don't share data with others can be are mapped to cores in a balanced manner. Both ways of mapping mentioned earlier improve balanced data locality, which, in its turn, saves energy.

The paper is structured as follows: the background and related work are presented in Section 2, then NUMA-BTLP algorithm [3] is presented in Section 3, followed by the ways of mapping threads, proposed by NUMA-BTDM [3] algorithm, in Section 4, while in Section 5, the materials and methods for measuring the power consumption are presented, followed by the experimental results in Section 6, discussions in Section 7 and conclusions, in the last section.

2. Background and related work

The purpose of a JIT compiler is to compile code when it is needed, and not compile the entire program to the disk as a compiler commonly does [1]. This is done by making the LLVM IR code executable in the context of the JIT process and by adding it to execution using the `addModule` function call [1]. Next, a symbol table is constructed and by using the `lookup` function call [1], every symbol in the table that has been not yet compiled, it is compiled. JIT can keep the optimizations that a common compilation would do, by creating a `llvm::FunctionPassManager` instance and configuring it with a set of optimizations [1]. Then, `PassManager` is run on the on a `Module` to get a more optimized form while keeping the semantics [1]. To optimize the code, JIT process uses an `optimizeModule` function call. Reaching the code is done by calling an `add` method on an `OptimizeLayer` object instead of calling `addModule` function [1]. The drawback of using LLVM JIT is that LLVM needs to be installed and compiled on the same machine as the one code needs to be run on.

Usually, JIT compilation is used to compile new features of a language that would otherwise not be compiled as usual during the compilation process, but rather by the JIT, before runtime. Instantiation of a language construct can be also JIT compiled. Paper [6] uses LLVM to build a JIT. In paper [6], the function templates are instantiated at runtime by providing non-constant expressions to the non-type template parameters and strings from which the type is deduced by conversion to type template parameters. Another paper [7] allows multiple definitions for the same language construct, producing different Abstract Syntax Trees (AST). While LLVM Clang is used to separate the unique AST from the redefined ones, the redefined AST is JIT compiled [7]. JIT compiled code can be also optimized [8].

Unlike in this present paper, where insertion of code is done in the LLVM IR, another paper would perform the insertion in the machine code [9]. The code that is inserted is delimitated by a region. Changing and deletion of the machine code is done in a similar way using delimitating regions [9]. The approach in paper [9] increases flexibility. However, in this paper, the same LLVM IR code is added anywhere in the input code. Therefore, this is not affecting the flexibility of the design.

LLVM JIT Framework is used to construct rules written in LLVM IR that would match to the language constructions from the input file. The process is called JIT compilation of the input file. Therefore, many times JIT compilers are used for compiling programming languages that are invented by the writer.

The custom JIT compiler in this paper does not create any construct rule, instead, the JIT compiler is able to create a `llvm::FunctionPassManager` instance [1] and to add to it the NUMA-BTLP [3] optimization pass. This way, the JIT compiler avoids compiling all the code for applying the optimization pass, before each time an application is first run on a machine.

The advantage of NUMA-BTLP [3] being a static optimization for mapping threads is that, regardless of the number of threads that are created, every thread is mapped considering the communication with the overall pool of threads. NUMA-BTLP [3] detects the LLVM IR [1] that creates the thread, and adds after each such call, another call that maps thread. Paper [10] disagrees with having a static thread mapping, and presents a mechanism for shared memory, through which the operating system can map the threads at runtime based on the communication between threads. However, mapping a thread in paper [10] does not consider the communication of the thread with all other threads, but rather the communication of the thread with another sharing the same data [10], which reduces the replication of data in different caches, thereby increasing the cache space available to the application [11]. Postponed threads are one of the type of threads proposed by the thread classification of NUMA-BTLP [3], which are mapped on the less loaded core. This paper supports the idea that postponed threads assure the balance of the computations and that they are mapped depending on all other threads, which cannot be guaranteed if mapping is performed at runtime, since not all other threads are mapped when the postponed thread is to be mapped.

Along with thread mapping, paper [12] also uses data mapping to minimize the number of RAM memory fetches. Paper [13] analyses many other aspects besides RAM memory fetches, including “L1 I/D-caches, I/D TLBs, L2 caches, hardware prefetchers, off-chip memory interconnects, branch predictors, memory disambiguation units and the cores” to determine the interaction between threads. Another research [14] concludes that having threads executed close in term of NUMA distance to where the main thread is executed is better than having threads executed on cores far from the core that executes the main thread, since it improves data locality. However, research in [15] states that locality-based policies can lead to performance reduction when communication is imbalanced.

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 26 June 2023 doi:10.20944/preprints202306.1834.v1

3. NUMA-BTLP Algorithm

3.1. Data dependencies considerations

To find the type of the thread, NUMA-BTLP algorithm [3] searches for data dependencies between each thread in the communication tree added so far and the thread created by the `pthread_create` call [5]. To get the data dependencies between a thread that is candidate to be added in the communication tree and a thread that is already added in the tree, a static analysis of the data used in the function attached to a thread from the communication tree and the data used by the function attached to the candidate thread, is performed [4].

An autonomous thread does not have any data dependencies with any other thread [2]. Such type of thread does not read any data written by other threads, does not write any data read by other threads and can only read data read by another thread [4].

A thread `i` is side-by-side relative to `j` if the two threads have data dependencies [4].

A postponed thread is in side-by-side relation only with the generation thread [4].

3.2. Data structures used in the implementation

The algorithm uses two tree data structures in its implementation that contain relevant information for the mapping: a thread generation tree and a thread communication tree. The tread generation tree is constructed as follows:

1. The main thread which executes the main function is the root of the generation tree and forms the first level of the tree
2. Threads that are created in the main function via `pthread_create` call [5] are sons of the root, forming the second level on the tree
3. The threads that are created in the functions executed by the treads (so called attached functions) in the second level form the third level in the tree and so on until the last level is formed.

Therefore, if a thread creates another thread the first tread will be the parent of the second in the generation tree.

The communication tree defines the data dependencies between the treads each corresponding to one or multiple nodes in the communication tree and it is constructed ed on the following rules:

1. The main thread which executes the main function id the root of the communication tree and forms the first level of the tree
2. If the execution thread that is candidate to be added in the communication tree is determined to be autonomous or postponed, the candidate is added in the communication tree as son threads of his parent from the generation tree
3. If the execution thread is side-by-side, the candidate is added as son thread of all the threads added already in the communication tree, with which the candidate is in side-by-side relation.

4. Mapping of NUMA-BTDM Algorithm

NUMA-BTDM algorithm [2] maps the threads to cores depending on their type and on the hardware details of the underlying machine. Mapping is performed by traversing the communication tree and assigning a core to each thread in the tree [4]. NUMA-BTDM algorithm [2] performs a logical mapping [4] by inserting a `pthread_setaffinity_np` call [5] in the LLVM IR of the input code, after each `pthread_create` call [5]. The `pthread_setaffinity_np` call [5] maps the thread to one or several cores. However, the real mapping is done at runtime, by calling the `pthread_setaffinity_np` call [5].

4.1. Getting the hardware architecture details of the underlying architecture

To obtain the hardware architecture detail to customize the mapping, NUMA-BTDM algorithm [2] executes twice the `system` call from inside LLVM IR code. First `system` call gives the number of cores of the underlying architecture and second `system` call gives the total number of logical cores. Dividing the number of logical cores to the number of cores gives the number of logical cores per CPU, used by the custom mapping in the NUMA-BTDM algorithm [2], assuming the architecture is homogeneous.

Physical number of cores is obtained by executing from the code of NUMA-BTLP algorithms [3], the system calls with the following parameter: system("cat /proc/cpuinfo | awk '/^processor/{print $3}' | wc -l; grep -c ^processor /proc/cpuinfo"). The logical number of cores is obtained by executing the system with the following parameter: system("grep ^cpu\\scores /proc/cpuinfo | uniq | awk '{print $4}'"). The /proc/cpuinfo file in Linux is written at every system start-up and contains relevant hardware information about the machine. By dividing the result of the second system call to the result of the first, the logical number of cores per processing unit is obtained. The number of cores and the number of logical cores per processing unit are used in the mapping of NUMA-BTDM algorithm [2].

4.2. Mapping autonomous threads

To get the autonomous threads, the communication tree is traversed, and the autonomous threads are stored in list. Then, the thread is mapped uniformly processing units in the following manner:

$$\begin{gathered}\text{coreIDForThreadi=floor((k×(i-1)) modulo j)}\\ \text{where i=1→threadNo, threadNo=total number of autonomous threads}\\ \text{and k = (threadNo (total number of autonomous threads))/(coreNo (total number of cores))}\end{gathered}\tag{1}$$

Given the thread ID for an autonomous thread, the information is stored in an unordered_map defined by (key, value) pairs, in which the thread ID is the key and the ID of the core to which the thread is mapped, is the value. Then, the LLVM IR input code is parsed and on every `pthread_create` call [5] found, which creates an autonomous thread, a `pthread_setaffinity_np` call [5] is inserted after it, mapping the autonomous thread to its core by using the information in the unordered_map. The corresponding core of the autonomous thread is quickly accessed due to the hash-based implementation of the unordered_map that stores the pairs (threads ID, core ID).

4.3. Mapping side-by-side threads

The communication tree is traversed searching for side-by-side threads. Once a side-by-side thread is found, it is added in an unordered_map similar as for autonomous threads. Once all side-by-side threads are added in the unordered_map, the code is parsed to reach the `pthread_create` calls [5] corresponding to each side-by-side thread. The call is reached uniquely by the thread ID, by the containing function i.e. called parent function and by the executing function i.e. called function attached to the thread, information that has been stored inside the tree node.

Every node in the communication tree corresponds to a `pthread_create` call [5] that creates a thread. Anyway, multiple nodes in the communication tree can define the same thread. A shared flag between multiple nodes in the tree that represent the same thread is set to “mapped” in the tree node once the first side-by-side thread e.g. that is created enclosed in a loop structure, is mapped. Therefore, the static analysis of NUMA-BTLP algorithm [3] considers all the threads created in a loop to all have the same mapping [4]. Anyway, the static analysis considers that the `pthread_create call` [5] in a loop maps a single thread because the compiler parses only a `pthread_create` call [5] by the static point of view [4]. So, the relation between the affinity of the mapped thread(s) and the `pthread_create` call [5] is one-to-one.

Another way of having multiple nodes in the communication tree defining the same thread is when the same `pthread_create` call [5] is called from another code path. In the static analysis of NUMA-BTLP algorithm [3], two or many nodes in the communication tree can each describe the same thread meaning the thread has the same ID, the same parent function and function attached in multiple nodes, when the parent function is called from different paths in the LLVM IR code. In this case, the static analysis [3] considers that the `pthread_create` call [5] creates the same thread from whichever path is called.

There is also another way of having multiple nodes in the communication tree pointing to the same thread. The same `pthread_create` call [5] can be represented by multiple nodes in the tree showing the thread created by the Pthreads call [5] has different data dependencies with different nodes from the communication tree. For instance, a thread that is side-by-side related to two threads will be the son of each of the two threads in the communication tree. Let's discuss this situation in the following. The `pthread_create` calls [5] corresponding to the son thread and to the two parent threads in the generation tree are reached by parsing the LLVM IR code and data dependencies are found between the son thread and the two other threads so called parent threads. Finally, the two parent threads will determine the son threads to be added in the communication tree in two different places, as son of the first parent thread and as son of the second parent thread. Once having reached the `pthread_create` call [5] of the son thread, the `pthread_setaffinity_np` calls [5] for the son thread and for the parent threads are inserted in the LLVM IR in the following way. Son thread is added to the communication tree as son of the first parent thread, son thread reunites its affinity with the affinity of the first parent thread, resulting its new affinity, son thread is added to the second parent thread in the communication tree, it reunites its newly computed affinity (computed as son of the first thread) with the affinity of the second parent resulting a new affinity for the son threads which is updated automatically in the son node from the first and the second parent thread. A good example for such a case is a data dependency between two threads created earlier in the parent function and another thread created afterwards in the same parent function which depends on the data used by the two threads.

In Figure 1 there is a thread with ID 0 which is the main thread executing the main function and the threads with IDs 1 and 2 are created using `pthread_create` calls [5] from inside the main function. The figure shows the communication tree for the threads described earlier. Given that thread 0 has data dependencies with thread 1, thread 1 has data dependencies with thread 0 and thread 2, all threads in the figure are of type side-by-side. According to earlier mentions in the paragraph below, thread 1 appears in two places in the communication tree because it is in side-by-side relation with two threads, namely 0 and 2. Thread with ID 0 is mapped on core 0 and thread with ID 2 is mapped on core 1, so the son thread with ID 1 of both threads with ID 0 and 2 will be mapped on cores 0 and 1, therefore, taking the affinity of both parent threads.

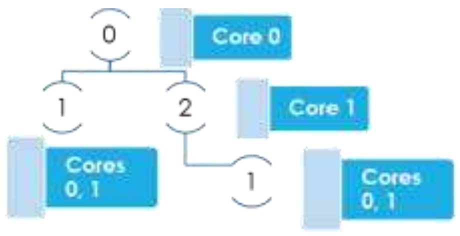

Figure 1. Mapping of a side-by-side thread having multiple parents in the communication tree.

4.4. Mapping postponed threads

Postponed threads are mapped each to the less loaded core [4], after mapping the autonomous and the side-by-side-threads. The less loaded core is considered the one that has the lowest number of threads assigned to it, at the specific moment. To find the less loaded core, the unordered_map is traversed and the (key, value) pairs are reversed and added in a map. Therefore, the (key, value) pairs in the unordered_map will become (new_key, new_value) in the map, where the new_key is a value in the unordered_map (i.e. value becomes thread ID) and the new_value is the key from the unordered_map (i.e. key becomes the list of core IDs where the thread is mapped). map container is sorted ascending by the number of threads assigned to each core. The postponed threads will always be mapped, one by one, to the first core in the map. However, if the first core changes, the postponed threads will be mapped to the new first core in the map.

5. Materials and Methods

To obtain the experimental results, power consumption has been measured for several real benchmarks, namely cpu-x, cpu, flops and context switch. Power consumption gain for each real benchmark was obtained by subtracting the power consumption measured without applying the NUMA-BTLP algorithm [3] to the real benchmark and the power consumption measured when the algorithm was applied to the benchmark [4]. Power consumption measurements above were obtained both on a NUMA system, which has a complex memory hierarchy and on a UMA system which has a single shared memory. The same system was configured as NUMA and the as UMA, by accessing its BIOS. So, the following measurements were done on both UMA and NUMA and with and without applying the NUMA-BTLP algorithm [3]. Each real benchmark except cpu-x which runs in infinite loop, was run 40 times and the average execution time of the 40 rounds was obtained [4]. The real benchmarks were also run 40 times to get the average power consumption both on the CPU and on the entire system [4]. CPU power consumption was measured using turbostat tool, which is a Linux program, and system power consumption was measured with physical device WattsUp [4]. The rate for obtaining both measures was 1 second [4]. Real benchmarks running for more than 15 minutes were stopped by using timeout system call [4]. The minimum, the maximum, the average, the variance and the standard deviation for the 1 second rate measurements of each of the 40 rounds, were computed [4]. The experimental results on the benchmarks were obtained by manually inserting `pthread_setaffinity_np` calls [5] in the code of the real benchmarks, according to the rules of the thread classification and mapping algorithms.

6. Results

Figure 2 shows for each real benchmark the power consumption gain in percentages by optimizing using the NUMA-BTLP algorithm [3], for both UMA and NUMA system, measured on a 32 core Fujitsu machine, configured with two NUMA nodes, each of 16 cores.

The cpu-x real benchmark run is infinite loop with less than 12 side-by-side that can be configured at runtime by the user. As expected, side-by-side threads produce the biggest optimization since the side-by-side threads use the same data which is kept in the cache avoiding memory fetch operations.

cpu benchmark runs with 16 autonomous threads. The optimization for cpu is lower as in case of cpu-x, because each autonomous thread is mapped on a separate core, so the threads fetch the required data in the L1 cache 16 times, or more as required. However, the difference between the optimization on UMA and the optimization on NUMA is lower in case of cpu benchmark, showing that fetches from a main memory (UMA) are more expensive than from the L1 cache (NUMA) as the number of fetches increases, in case of autonomous threads.

flops real benchmark runs with 2400 autonomous threads. The optimization is lower as in case of cpu-x and cpu because the thread overhead is bigger in case of flops. However, flops benchmark is optimized more on NUMA, showing that as the number of threads increases the bigger is the optimization using the algorithms, on NUMA systems.

context switch runs with 2 autonomous threads. The optimization is low because the number of threads is small. The algorithms in this paper prove to perform better on medium number of threads, preferably side-by-side threads in majority.

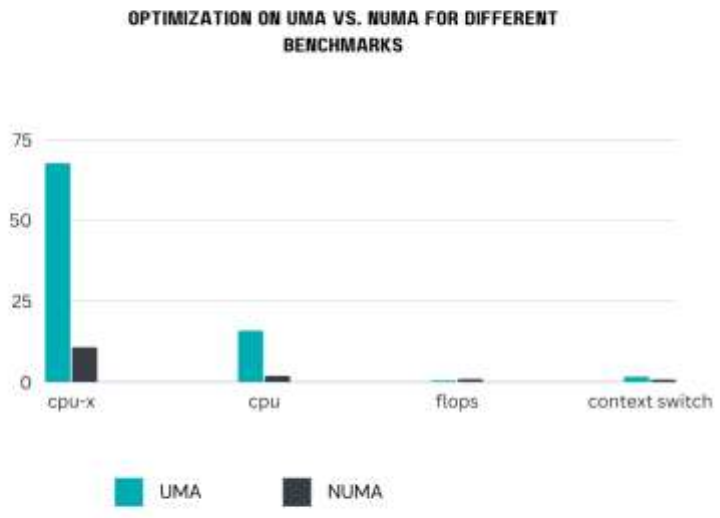

Figure 2. Experimental results of energy optimization percentage on different real benchmarks

Table 1 shows the power gain (by applying NUMA-BTLP [3]) for each benchmark in Watt/s and in percentages, on both UMA and NUMA systems.

Table 1. Power gain in Watt/s and in percentages when applying NUMA-BTLP [3] to each benchmark that is run on both UMA and NUMA systems

Real Benchmark	Optimization in W/s		Optimization in %	
	UMA	NUMA	UMA	NUMA
cpu-x	1.29	0.19	67.63	10.75
cpu	7.56	0.9	15.79	1.88
flops	0.3	0.6	0.41	0.82
context switch	0.77	0.32	1.57	0.76

7. Discussion

NUMA-BTDM [2] is a static mapping algorithm applied to parallel applications using Pthreads library [5] for spawning the threads. The algorithm decides the CPU affinity of each thread based on its type [3]. The type of the thread is assigned based by NUMA-BTLP algorithm [3]. The algorithm classifies the execution treads into autonomous, side-by-side and postponed based on the static data dependencies between threads [4]. Both NUMA-BTDM [2] and NUMA-BTLP [3] algorithms contribute to better balanced data locality on NUMA systems by optimizing the mapping of threads on the systems [4]. Moreover, the energy consumption is optimized [4]. NUMA-BTDM [2] uses Pthreads library [5] for setting the CPU affinity, enabling better proximity in time and NUMA distance threads and the data they use [4]. The novelties of tis paper consist of the following:

- The ability to allow the parallel applications written in C that use Pthreads library [5] to customize and control the thread mapping based on static characteristics of the code by inserting `pthread_setaffinity_np` calls [5] in the LLVM IR of the input code after each `pthread_create` call [5]. Thus, the mapping is not random
- The disadvantage of not knowing the number of the execution threads at compile-time is eliminated by inserting `pthread_setaffinity_np` calls [5] in the LLVM IR of the input code after each `pthread_create` call [5], allowing all threads to by mapped regardless of their number
- Paper defines original static criteria for classifying threads in 3 categories and defines the categories

- Mapping of threads depending on their type. The autonomous threads are distributed uniformly on cores allowing better balance in achieving balanced data locality. A side-by-side tread is mapped on the same cores as each other thread with respect to which it is considered side-by-side allowing better data locality [4]
- The definition of the static criteria of classifying the execution threads in three categories and the classification itself. If two threads are data dependent (i.e. the data send to a thread execution is used in the execution of the other thread), they are classified as side-by-side [4]. If a thread has no data dependencies with any other thread, the thread is of type autonomous [4]. If a thread has data dependencies with its parent thread only, the thread type is postponed [4]. The data dependencies are revealed by NUMA-BTLP algorithm [3] which is implemented in LLVM, but not yet part of it
- Mapping execution threads based on their type. The execution of autonomous threads is spread uniformly to cores, which ensures the completion of the balance criteria in achieving the balanced data locality [4]. A side-by-side thread is allocated for execution on each of the cores on which the threads in relation of side-by-side to the thread, are mapped. The previous ensures achieving optimized data locality [4]. The postponed threads are mapped to the less loaded core so far, once they are identified in the traversing the communication tree. The distribution of postponed threads ensures also balanced execution as the distribution of autonomous threads
- Integrating the implementation of the classification and mapping algorithms in a modern compiling infrastructure such as LLVM
- Using two trees, a generation and a communication tree in mapping the execution threads. The communication tree describes the data dependencies between threads and the generation tree describes the generation of the execution threads [4]. The way in which the communication tree is constructed represents novelty. The rules of constructing the tree are the following: any autonomous or postponed thread is added as a son thread to every occurrence in the communication of its parent in the generation tree and every side-by-side thread is added as a son thread to every thread with which he is in side-by-side relation. By constructing the communication tree in the above manner, one can find out the way threads are communicating, by traversing the tree.

8. Conclusions

In this paper was presented a prototype of a LLVM JIT. The JIT can turn the native code of a C/C++ application that uses Pthreads [5], into corresponding LLVM IR [1], is able to call the NUMA-BTLP optimization pass which adds the Pthreads mapping calls [5] into the LLVM IR. Finally, the LLVM IR is converted back to native code.

NUMA-BTDM [2] (called by NUMA-BTLP algorithm), is among the few static thread mapping optimization designed for NUMA systems. The NUMA-BTDM [2] and NUMA-BTLP [3] algorithms improve the balanced data locality in an innovative manner [16]. The algorithms map autonomous threads to cores so that the overall execution of the application is balanced [16]. The previous avoids loading one core with the execution of multiple threads, while the other cores haven't got any thread to execute. Another novelty of the two algorithms is that they ensure the proximity in time and NUMA distance of the execution of side-by-side threads to the data they use [16]. Furthermore, the postponed threads do not steal the cache from the other threads since they are mapped on the less loaded cores [16].

The algorithms in this paper improve the power consumption for a small number of autonomous execution threads with 2%, for a small number of side-by-side threads with 15%, for a medium number of autonomous threads with 1% and the algorithms do not degrade the execution time [16].

Balanced data locality is obtained by sharing the same L1 cache by threads and by distributing the threads uniformly to cores [16]. Moreover, the energy spent with the interconnections is reduced when the number of execution threads is medium [16].

Author Contributions: For research articles with several authors, a short paragraph specifying their individual contributions must be provided. The following statements should be used "Conceptualization, I.S and G.R.G.; methodology, I.S.; software, I.S.; validation, I.S.; formal analysis, I.S.; investigation, I.S.; resources, I.S.; data curation, I.S.; writing—original draft preparation, I.S. and G.R.G.; writing—review and editing, I.S.; visualization, I.S.; supervision; project administration, I.S.; funding acquisition. All authors have read and agreed to the published version of the manuscript." Please turn to the CRediT taxonomy for the term explanation.

Funding: This research received no external funding.

Data Availability Statement: The data presented in this study are available in Iulia Știrb, "Reducing the energy consumption and execution time through the optimization of the communication between threads and through balanced data locality at the execution of parallel programs, on NUMA systems", PhD Thesis

Conflicts of Interest: The authors declare no conflict of interest.

References

1. LLVM Compiler Infrastructure Project, Available online: https://llvm.org/ (accessed: 4th of February 2021)
2. Ştirb, I., NUMA-BTDM: A thread mapping algorithm for balanced data locality on NUMA systems. In *2016* 17th International Conference on Parallel and Distributed Computing, Applications and Technologies (PDCAT), December 2016, pp. 317-320.
3. Ştirb, I., NUMA-BTLP: A static algorithm for thread classification. In 2018 5th International Conference on Control, Decision and Information Technologies (CoDIT), April 2018, pp. 882-887.
4. Știrb, I., 2018. Extending NUMA-BTLP Algorithm with Thread Mapping Based on a Communication Tree. *Computers*, *7*(4), p.66.
5. pthreads(7) - Linux manual page, Available online: https://man7.org/linux/man-pages/man7/pthreads.7.html (accessed: 4th of February 2021)
6. Finkel, H., Poliakoff, D., Camier, J.S. and Richards, D.F., Clangjit: Enhancing c++ with just-in-time compilation. In 2019 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC), November 2019, pp. 82-95.
7. López-Gómez, J., Fernández, J., Astorga, D.D.R., Vassilev, V., Naumann, A. and García, J.D., Relaxing the one definition rule in interpreted C++. In Proceedings of the 29th International Conference on Compiler Construction, February 2020, pp. 212-222.
8. Auler, R. and Borin, E., 2013. A LLVM Just-in-Time Compilation Cost Analysis. Technical Report 13-2013 IC-UNICAMP.
9. Ansel, J., Marchenko, P., Erlingsson, U., Taylor, E., Chen, B., Schuff, D.L., Sehr, D., Biffle, C.L. and Yee, B., Language-independent sandboxing of just-in-time compilation and self-modifying code. In Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation, June 2011, pp. 355-366.
10. Diener, M., Cruz, E.H., Navaux, P.O., Busse, A. and Heiß, H.U., 2015. Communication-aware process and thread mapping using online communication detection. *Parallel Computing* **2015**, *43*, pp.43-63.
11. Chishti Z., Powell M. D., Vijaykumar T. N., Optimizing Replication, Communication, and Capacity Allocation in CMPs. In *ACM SIGARCH Computer Architecture News*, 2005, Volume 33 (2), pp. 357–368. doi:10.1145/1080695.1070001
12. Cruz, E.H., Diener, M., Pilla, L.L. and Navaux, P.O., Hardware-assisted thread and data mapping in hierarchical multicore architectures. *ACM Transactions on Architecture and Code Optimization (TACO)*, *13*(3), 2016, pp.1-28.
13. Wang, W., Dey, T., Mars, J., Tang, L., Davidson, J.W. and Soffa, M.L., Performance analysis of thread mappings with a holistic view of the hardware resources. In 2012 IEEE International Symposium on Performance Analysis of Systems & Software, April 2012, pp. 156-167.
14. Mallach, S. and Gutwenger, C., Improved scalability by using hardware-aware thread affinities. In *Facing the multicore-challenge: aspects of new paradigms and technologies in parallel computing*, 2010, pp.29-41.
15. Diener, M., Cruz, E.H., Alves, M.A., Alhakeem, M.S., Navaux, P.O. and Heiß, H.U., Locality and balance for communication-aware thread mapping in multicore systems. In *Euro-Par 2015: Parallel Processing: 21st International Conference on Parallel and Distributed Computing, Vienna, Austria, August 24-28, 2015, Proceedings 21*, 2015, pp. 196-208.
16. Știrb, I. Reducerea consumului de energie și a timpului de execuție prin optimizarea comunicării între firele de execuție și prin localizarea echilibrată a datelor la execuția programelor paralele, pe sisteme NUMA, Politehnica University of Timișoara, Timișoara, 9th of December 2020