Article

A Low-Level Virtual Machine Just-In-Time Prototype for Running an Energy-Saving Hardware-Aware Mapping Algorithm on C/C++ Applications That Use Pthreads

Iulia Știrb 1,* and Gilbert-Rainer Gillich 1,2

1 Department of Engineering Science, Babes-Bolyai University, P-ta Traian Vuia 1-4, 320085 Resita, Romania; gilbert.gillich@ubbcluj.ro
2 Doctoral School of Engineering, Babes-Bolyai University, P-ta Traian Vuia 1-4, 320085 Resita, Romania
* 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 a clang compiler and its elaborated middle end containing different optimizations that improve the runtime performance. This paper specifically addresses the part of building a JIT compiler using an LLVM with the scope of obtaining 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 the NUMA-BTLP static thread classification algorithm and to the NUMA-BTDM static thread mapping algorithm. Afterwards, the hardware-aware algorithms are run using the JIT compiler within an optimization pass. The 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 achieve this, the JIT compiler takes the native code of the application, obtains 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 by up to 15% on the 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 the LLVM IR (Intermediate Representation) corresponding to new syntactic rules as part of a new programming language invented by the developer. In this case, the LLVM JIT is used to compile the code which is not part of the language definitions of clang (LLVM front end). LLVM JIT [1] can also trigger LLVM optimizations on the LLVM IR. The JIT compiler prototype in this paper converts the native code into the LLVM IR using LLVM On-Request-Compilation (ORC) JIT API and applies the LLVM optimization pass composed of NUMA-BTDM [2] and NUMA-BTLP [3] to the LLVM IR. The NUMA-BTLP [3] optimization pass obtains the hardware details of the underlying architecture (the number of physical cores and the number of logical cores). Afterwards, the NUMA-BTLP algorithm [3] builds a communication tree that describes the data dependencies between threads and calls the NUMA-BTDM algorithm [2] that performs the hardware-aware thread mapping based on the communication between threads. After the optimization pass is applied to the LLVM IR, the LLVM IR is converted back to native code using the LLVM ORC JIT API.
The NUMA-BTLP [3] is a static thread classification algorithm implemented using an LLVM [1]. The algorithm is defined following three types of threads depending on the following static criteria [2]:

(1) autonomous threads—these threads have no data dependency with other threads;
(2) side-by-side threads—this type of thread has data dependencies with other threads;
(3) postponed threads—they only have data dependencies with the generating thread. The algorithm associates a thread with a tree node and adds the thread to a communication tree. The communication tree defines the way threads communicate with each other via their position in the tree, e.g., if thread \(i\) has data dependencies with thread \(j\), then thread \(i\) could be the parent of thread \(j\) in the communication tree or vice versa. Afterwards, the communication tree is traversed and each thread is mapped depending on its type (the type 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 the NUMA-BTDM algorithm [2] (implemented using an LLVM [1]), e.g., autonomous threads, which 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 they share data, while postponed threads are mapped on the less loaded processing unit at the time it is mapped (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 has to be Pthreads [5]. That is because the NUMA-BTLP algorithm [3] searches for pthread_create calls [5] (which creates the threads) in the LLVM IR [1] representation of the input code. One of the parameters of the call is the function executed by the thread. The NUMA-BTLP algorithm [3] performs a static data dependency analysis on the function, resulting in the thread type (autonomous, side-by-side or postponed) [4]. Afterwards, the thread is added in the communication tree based on its type.

Since the NUMA-BTLP [3] and NUMA-BTDM [2] optimization algorithms are implemented in an LLVM [1], the hardware details are obtained at compile time, resulting in a native code optimized for the same machine where it was compiled. Therefore, the optimization would not have any effect on other machines. Our solution to this is constructing a JIT compiler with an LLVM [1] that triggers the thread classification and the thread mapping optimization pass any time the application is first run on a machine. This way, the native code is optimized based on the hardware details of the underlying machine.

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

In previous research [2–4], it has been shown how the NUMA-BTLP [3] and NUMA-BTDM [2] algorithms improve energy consumption by inserting a Pthreads thread mapping call after each Pthreads thread creation call, which optimized the mapping. The mapping call 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 the Pthreads parallel model.

The advantages of using the hardware-aware mapping in [4] is that the threads that share data are mapped to the same cores and the threads that do not share data with others are mapped in a balanced manner. Both ways of mapping improve balanced data locality, which, in turn, saves energy.

This paper is structured as follows: the background and related work are presented in Section 2, then the NUMA-BTLP algorithm [3] is presented in Section 3, followed by the ways of mapping threads proposed with the 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

Usually, an LLVM On-Request-Compilation (ORC) JIT API is an API used to compile syntactic rules that are invented by software developers. One can develop a compiler for a language they invented and obtain the native code corresponding to the high-level code. The native code is compiled for a specific machine. Through LLVM JIT compilation, the LLVM IR corresponding to the native code can be compiled for another target. Therefore, an LLVM JIT API is useful for changing the target machine for which the high-level code was compiled. However, an LLVM JIT may have other uses.

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

Usually, JIT compilation is used to compile new features of a language, supported by clang, that would otherwise not be compiled as usual during the compilation process, but rather by the JIT compiler before runtime. The instantiation of a language construct can also be JIT-compiled. Paper [6] uses an LLVM to build a JIT. Paper [6] uses an LLVM to build a JIT compiler that provides a C++ language extension allowing template-based specialization to occur during program execution. Another paper [7] allows for multiple definitions of the same language construct, producing different Abstract Syntax Trees (ASTs). While an LLVM clang is used to separate the unique AST from the redefined ones, the redefined ASTs are JIT-compiled [7]. The JIT-compiled code can also be optimized [8].

Unlike in the present paper, where the insertion of code is completed in the LLVM IR, another paper performs the insertion in the machine code [9]. The code that is inserted is delimited by a region. The changing and deletion of the machine code is achieved 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 does not affect the flexibility of the design.

LLVM JIT Framework is used to implement syntactic rules written in an LLVM IR that would match the language from the input high-level file. The process is called the JIT compilation of the input file. Therefore, JIT compilers are often used for compiling programming languages that are invented by the JIT compiler developer.

The custom JIT compiler in this paper does not parse any syntactic rules; instead, the JIT compiler is able to create an llvm::FunctionPassManager instance [1] and add it to 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 LLVM JIT compiler in this paper is used to obtain the hardware details of the underlying machine (the number of physical and logical cores) and to apply the NUMA-BTLP optimization pass to the LLVM IR based on the hardware details obtained. The LLVM IR is obtained from the native code using the LLVM ORC JIT compilation. After the LLVM IR code is obtained, our LLVM JIT compiler creates an llvm::FunctionPassManager instance [1] and adds it to the NUMA-BTLP [3] optimization pass.

Compilation wastes time and resources. The advantage of using our LLVM JIT compiler is that the code is not compiled to obtain the LLVM IR. The LLVM IR is obtained very quickly from the native code using the LLVM ORC JIT API. Another advantage of using our approach is that the NUMA-BTLP optimization pass can be applied to the LLVM IR that
has been obtained from the native code. There is no need to compile the high-level code with clang to apply the optimization pass. The hardware details of the underlying machine are obtained using our approach of an LLVM JIT and the thread mapping is completed by the optimization pass based on the hardware details. Thread mapping is performed at the LLVM IR level. Afterwards, the LLVM IR is compiled back to the native code. The steps above are completed before the application is run for the first time on a machine. The advantage of using this approach is that the application is optimized for the underlying machine. The disadvantage of our approach is that the LLVM compiler needs to be built or installed on the machine for which the native code is optimized.

The advantage of the NUMA-BTLP [3] being a static optimization tool 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. The NUMA-BTLP [3] detects the LLVM IR [1] that creates the thread, and adds after each such call another call that maps the thread. Paper [10] disagrees with having 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, which is detected from page faults. The paper addresses the parallel applications that use shared memory for communication. 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 the 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] analyzes 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 paper [14] concludes that having threads executed close in terms of the 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, the research in [15] states that locality-based policies can lead to performance reduction when communication is imbalanced.

As for the NUMA-BTLP algorithm [3], a thread is created using a pthread_create call which assigns a function given as a parameter to a thread ID. The meaning of the assignment is that the thread with the specified ID will execute the function at runtime. On the other hand, thread mapping is completed using a pthread_setaffinity_np call which assigns the cores on which the thread (given by the thread ID) will run. Both the thread ID and cpu affinity are given as parameters to pthread_setaffinity_np. The mapping is achieved through the NUMA-BTDM algorithm [2] by inserting pthread_setaffinity_np calls in the LLVM IR before compiling the LLVM IR back to the native code using our JIT compiler.

3. NUMA-BTLP Algorithm
3.1. Data Dependency Considerations

To find the type of the thread, the 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 obtain the data dependencies between a thread that is a candidate to be added to the communication tree and a thread that has already been added to 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 a type of thread does not have read–write or write–read data dependencies with any other thread and can only read data read by any other thread [4].

A thread \( i \) is a side-by-side relative to \( j \) if the two threads have read–write, write–read or write–write data dependencies [4].

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

3.2. Data Structures Used in Implementation

The algorithm uses two tree data structures in its implementation that contain relevant information for mapping: a thread generation tree and a thread communication tree. The thread 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 the 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 thread 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 with the following rules:

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

4. Mapping of the NUMA-BTDM Algorithm

The NUMA-BTDM algorithm [2] maps the threads to the 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]. The NUMA-BTDM algorithm [2] performs 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 completed at runtime by calling the pthread_setaffinity_np call [5].

4.1. Obtaining the Hardware Architecture Details of the Underlying Architecture

To obtain the hardware architecture detail to customize the mapping, the NUMA-BTDM algorithm [2] executes the system call twice from inside the LLVM IR code. The first system call gives the number of cores of the underlying architecture and the second system call gives the total number of logical cores. Dividing the number of logical cores by 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.

The physical number of cores is obtained by executing, from the code of the 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 by 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 the NUMA-BTDM algorithm [2].

4.2. Mapping Autonomous Threads

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

\[
\text{coreIDForThread}_i = \text{floor}((k \times (i - 1)) \mod j)
\]

where \( i = 1 \to \text{threadNo}, \text{threadNo} = \text{total number of autonomous threads} \)

\( k = \frac{\text{threadNo} \times \text{total number of autonomous threads}}{\text{coreNo} \times \text{total number of cores}} \)  

(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 found on every `pthread_create` call [5], 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 to an unordered_map similar to autonomous threads. Once all side-by-side threads are added to 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 using the containing function, i.e., the called parent function, and by the executing function, i.e., the called function attached to the thread, to reach the 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. Moreover, 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 the NUMA-BTLP algorithm [3] considers all the threads created in a loop to all have the same mapping [4]. Moreover, the static analysis considers that the `pthread_create` call [5] in a loop maps a single thread because the compiler only parses a `pthread_create` call [5] from 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 the 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 that 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 us discuss this situation. 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 so-called parent threads. Finally, the two parent threads will determine the son threads to be added to the communication tree in two different places, as a son of the first parent thread and as a son of the second parent thread. 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 to be inserted in the LLVM IR in the following way. The son thread is added to the communication tree as a son of the first parent thread. The son thread reunites its affinity with the affinity of the first parent thread, resulting in its new affinity. The son thread is added to the second parent thread in the communication tree. It reunites its newly computed affinity (computed as a 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 threads. 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, and thread 1 has data dependencies with thread 0 and thread 2, all threads in the figure are of the side-by-side type. According to the 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 the 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 with ID 1 having multiple parents in the communication tree.](image)

**4.4. Mapping Postponed Threads**

Postponed threads are each mapped 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 a specific moment. To find the less loaded core, the unordered_map is traversed and the (key, value) pairs are reversed and added to 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., the value becomes the thread ID) and the new_value is the key from the unordered_map (i.e., the key becomes the list of core IDs where the thread is mapped). The 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, the power consumption was measured for several real benchmarks, namely cpu-x, cpu, flops and context switch. The 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 was measured when the algorithm was applied to the benchmark [4]. The above power consumption measurements were obtained both on a NUMA system, which has a complex memory hierarchy, and on an UMA system, which has a single shared memory. The same system was configured as NUMA and then as UMA by accessing its BIOS. So, the following measurements were completed on both UMA and NUMA systems and with and without the application of the NUMA-BTLP algorithm [3]. Each real benchmark, except cpu-x which runs in an 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 obtain the average power consumption both on the CPU and on the entire system [4]. The CPU power consumption was measured using the turbostat tool, which is a Linux program, and the system power consumption was measured with the physical device WattsUp [4]. The rate for obtaining both measures was 1 s [4]. Real benchmarks running for more than 15 min were stopped by using the timeout system call [4]. The minimum, the maximum, the average, the variance and the standard deviation for the 1 s 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 through optimization using the NUMA-BTLP algorithm [3] for both UMA and NUMA systems. These were measured on a 12-core Fujitsu Workstation which has 2 Intel Xeon E5-2630 v2 ivy bridge processors, where each processor is a NUMA node and each NUMA node has six cores, with two logical threads running on each core. The architecture has two levels of private cache (6 × 32 KB 8-way set associative L1 instruction caches, 6 × 32 KB 8-way set associative L1 data caches, 6 × 256 KB 8-way set L2 associative caches) which favor the increasing cache hit rate of autonomous threads. The shared cache is the L3 cache (15 MB 20-way set associative L3 shared cache). Sharing a larger dimension cache (L3 instead of L2) allows the side-by-side threads to share more data residing in the cache.

The cpu-x real benchmark is run in an infinite loop in the form of a GUI that displays the hardware parameters and memory configuration of the underlying system, operating system parameters and parameters regarding the run of the cpu-x benchmark itself. The cpu-x can run with a maximum number of 12 threads that can be changed dynamically by the user. The implementation of the cpu-x benchmark includes a for loop which creates the threads inside it using the pthread_create call. Therefore, the threads are side-by-side, since they have write–write data dependencies, as they all execute the same function. The NUMA-BTLP algorithm [3] inserts a pthread_setaffinity_np call [1] inside the for loop after the pthread_create call [1]. The pthread_setaffinity_np call [1] maps all threads to the same cores by giving a non-variable affinity as a parameter of the call. As expected, the side-by-side threads produce the largest optimization, since the side-by-side threads use the same data which is kept in the cache, avoiding memory fetch operations.
Figure 2. Experimental results of energy optimization percentage on different real benchmarks.

The cpu creates the threads in two functions. The two functions are called from the main function. One of the functions computes the number of floating point operations per second from a number of 33 instructions. The instructions are executed by each thread from three groups of threads. The three groups have 1, 2 and 4 execution threads, respectively. Similarly, the other function computes the number of integer operations per second. The threads are created by the main thread (which executes the main function) and there is no data dependency between these threads. Therefore, the type of the threads is autonomous and the threads are distributed uniformly among the processing units by the NUMA-BTDM algorithm [2] in the following manner: in case the number of threads is one, the thread is mapped to core 0; in case the number of threads is two, the two threads are mapped one on core 0 and the other on core six and in case the number of threads is four, they are mapped on cores 0, three, six and nine, respectively. cpu benchmark optimization is lower than cpu-x benchmark optimization, because each autonomous thread is mapped on a separate core, so the threads fetch the required data in the L1 cache a number of times equal to the number of threads. However, the difference between the optimization on UMA and the optimization on NUMA is lower in case of the 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.

The flops benchmark ran for approximately 10 min. The benchmark creates four execution threads 600 times. Similar to the cpu, flops maps the execution threads to cores 0, three, six and nine, respectively. Therefore, 600 execution threads will be mapped on each of the four cores. The flops benchmark computes the number of floating point operations per second for 600 s. The optimization is lower than in the cpu-x and cpu because the thread overhead is larger in case of flops. However, the flops benchmark is optimized more on NUMA, showing that, as the number of threads increases, the optimization is larger using the algorithms on NUMA systems.

The context switch benchmark has two main functions that are run alternatively. Each of the main functions creates an execution thread. The two threads have no data dependencies, so they are considered autonomous and mapped on cores 0 and six, respectively. The optimization is low because the number of threads is small. The algorithms in this paper
are proven to perform better on a medium number of threads, preferably with a majority of side-by-side threads.

Table 1 shows the power gain (by applying the 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 the NUMA-BTLP [3] to each benchmark that is run on both UMA and NUMA systems.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Optimization in W/s</th>
<th>Optimization in %</th>
</tr>
</thead>
<tbody>
<tr>
<td>cpu-x</td>
<td>1.29</td>
<td>67.63</td>
</tr>
<tr>
<td>cpu</td>
<td>7.56</td>
<td>15.79</td>
</tr>
<tr>
<td>flops</td>
<td>0.3</td>
<td>0.41</td>
</tr>
<tr>
<td>context switch</td>
<td>0.77</td>
<td>1.57</td>
</tr>
</tbody>
</table>

7. Discussion

The NUMA-BTDM [2] is a static mapping algorithm applied to parallel applications using the Pthreads library [5] for spawning the threads. The algorithm decides the CPU affinity of each thread based on its type [3]. The type of thread is assigned based on the NUMA-BTLP algorithm [3]. The algorithm classifies the execution threads into autonomous, side-by-side and postponed based on the static data dependencies between threads [4]. Both the 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]. The NUMA-BTDM [2] uses the 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 this paper consist of the following:

- The ability to allow the parallel applications written in C that use the Pthreads library [5] to customize and control the thread mapping based on the 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 be mapped regardless of their number.
- This paper defines the original static criteria for classifying threads into three categories and defines these categories.
- The mapping of threads depends on their type. The autonomous threads are distributed uniformly on cores allowing for a better balance in achieving balanced data locality. A side-by-side threads are mapped on the same cores as each other with respect to which are considered side-by-side, allowing for better data locality [4].
- The definition of the static criteria for classifying the execution threads in three categories and the classification itself. If two threads are data-dependent (i.e., the data sent 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 type is autonomous [4]. If a thread only has data dependencies with its parent thread, the thread type is postponed [4]. The data dependencies are revealed by the NUMA-BTLP algorithm [3] which is implemented in an LLVM, but is not yet part of it.
- The mapping of execution threads is based on their type. The execution of autonomous threads is spread uniformly to the cores, which ensures the completion of the balance criteria in achieving balanced data locality [4]. A side-by-side thread is allocated for execution on each of the cores on which the threads in side-by-side relation to the thread are mapped. This ensures the achievement of optimized data locality [4]. The postponed threads are mapped to the less loaded core so far once they are identified...
in the traversing of the communication tree. The distribution of postponed threads also ensures the balanced execution of the distribution of autonomous threads.

- The implementation of the classification and mapping algorithms was integrated in a modern compiling infrastructure such as LLVM.
- Two trees, a generation and a communication tree, were used 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 to which it 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.

Table 2 presents a comparison of state-of-the-art mapping algorithms from the point of view of representing the communication between execution threads.

Table 2. Comparison of state-of-the-art mapping algorithms from the point of view of representing the communication between execution threads.

<table>
<thead>
<tr>
<th>Mapping Algorithms</th>
<th>Execution Time Matrix</th>
<th>Communication Matrix</th>
<th>Task Graph</th>
<th>Communication Graph</th>
<th>Communication Tree</th>
<th>Hardware Architecture Graph</th>
<th>Hardware Architecture Tree</th>
</tr>
</thead>
<tbody>
<tr>
<td>Greedy algorithms [16]</td>
<td>✓</td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Scotch [17]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>METIS [18]</td>
<td></td>
<td></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Zoltan [19]</td>
<td></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TreeMatch [20]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EagerMap [21]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NUMA-BTLP [3]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 3 presents a comparison of state-of-the-art mapping algorithms from the point of view of the mapping technique used.

Table 3. Comparison of state-of-the-art mapping algorithms from the point of view of the mapping technique used.

<table>
<thead>
<tr>
<th>Mapping Algorithms</th>
<th>Execution Time Estimation (from History)</th>
<th>Greedy/Recursive Algorithms</th>
<th>(Bi)partition/Resize of the Matrix</th>
<th>Graph Expansion</th>
<th>Grouping of Tasks/Execution Threads</th>
<th>Reduction in the Height of the Hardware Architecture Tree</th>
<th>Mapping Based on the Communication Tree</th>
</tr>
</thead>
<tbody>
<tr>
<td>OLB, LBA, Greedy algorithm [16]</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Scotch [17]</td>
<td></td>
<td>✓</td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>METIS [18]</td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Zoltan [19]</td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TreeMatch [20]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EagerMap [21]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NUMA-BTLP [3]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 4 presents a comparison of state-of-the-art mapping algorithms from the point of view of the moment of mapping.
the thread type is autonomous [4]. If a thread only has data dependencies with its
parent thread, the thread type is postponed [4]. The data dependencies are revealed
in the above manner, one can
is added as a son thread to every occurrence in the communication of its parent in
rules of constructing the tree are the following: any autonomous or postponed thread
threads and the generation tree describes the generation of the execution threads [4].

cated for execution on each of the cores on which the threads in side-by-side relation
balance criteria in achieving balanced data locality [4]. A side-by-side thread is allo-
the thread are mapped. This ensures the achievement of optimized data locality

ted for execution on each of the cores on which the threads in side-by-side relation

In the following, we present a brief conclusion regarding the comparison between our mapping algorithm and other mapping algorithms. Scotch [17], which is also a static mapping algorithm like our algorithms, optimizes the energy consumption by up to 20%, while our static mapping algorithm optimizes the energy consumption by up to 15% for tested real benchmarks. Static mapping algorithms avoid the overhead with runtime mapping. However, EagerMap [21], which is a dynamic mapping algorithm, executes 10 times faster than other dynamic mapping algorithms.

TreeMatch [20] is another mapping algorithm that, like our algorithm, aims to reduce the communication costs between threads.

8. Conclusions

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

The NUMA-BTDM [2] (called the NUMA-BTLIP algorithm), is among the few static thread mapping optimization tools designed for NUMA systems. The NUMA-BTDM [2] and NUMA-BTLIP [3] algorithms improve the balanced data locality in an innovative manner [22]. The algorithms map autonomous threads to cores so that the overall execution of the application is balanced [22]. This avoids loading one core with the execution of multiple threads, while the other cores do not have any threads 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 [22]. Furthermore, the postponed threads do not steal the cache from the other threads since they are mapped on the less loaded cores [22].

The algorithms in this paper improve the power consumption for a small number of autonomous execution threads by 2%, for a small number of side-by-side threads by
15% and for a medium number of autonomous threads by 1%, and the algorithms do not degrade the execution time [22].

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

**Author Contributions:** 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.; All authors have read and agreed to the published version of the manuscript.

**Funding:** This research received no external funding.

**Data Availability Statement:** The data presented in this study are available in Iulia Stirb, “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”, Ph.D. thesis.

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

**References**

4. Stirb, I. Extending NUMA-BTLP Algorithm with Thread Mapping Based on a Communication Tree. *Computers 2018*, 7, 66. [CrossRef]


Disclaimer/Publisher’s Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.