Hindawi
Wireless Communications and Mobile Computing
Volume 2021, Article ID 6611867, 12 pages
https://doi.org/10.1155/2021/6611867

Research Article

Enhancing Dynamic Binary Translation in Mobile Computing by Leveraging Polyhedral Optimization

Mingliang Li, Jianmin Pang, Feng Yue, Fudong Liu, Jun Wang, and Jie Tan

State Key Laboratory of Mathematical Engineering and Advanced Computing, Zhengzhou, Henan 450000, China

Correspondence should be addressed to Jianmin Pang; jianmin_pang@hotmail.com

Received 27 November 2020; Revised 9 March 2021; Accepted 5 April 2021; Published 19 April 2021

Academic Editor: Shaohua Wan

Dynamic binary translation (DBT) is gaining importance in mobile computing. Mobile Edge Computing (MEC) augments mobile devices with powerful servers, whereas edge servers and smartphones are usually based on heterogeneous architecture. To leverage high-performance resources on servers, code offloading is an ideal approach that relies on DBT. In addition, mobile devices equipped with multicore processors and GPU are becoming ubiquitous. Migrating x86_64 application binaries to mobile devices by using DBT can also make a contribution to providing various mobile applications, e.g., multimedia applications. However, the translation efficiency and overall performance of DBT for application migration are not satisfactory, because of runtime overhead and low quality of the translated code. Meanwhile, traditional DBT systems do not fully exploit the computational resources provided by multicore processors, especially when translating sequential guest applications. In this work, we focus on leveraging ubiquitous multicore processors to improve DBT performance by parallelizing sequential applications during translation. For that, we propose LLPEMU, a DBT framework that combines binary translation with polyhedral optimization. We investigate the obstacles of adapting existing polyhedral optimization in compilers to DBT and present a feasible method to overcome these issues. In addition, LLPEMU adopts static-dynamic combination to ensure that sequential binaries are parallelized while incurring low runtime overhead. Our evaluation results show that LLPEMU outperforms QEMU significantly on the PolyBench benchmark.

1. Introduction

A DBT system can run guest applications transparently on a host machine with a different Instruction Set Architecture (ISA), and DBT is gaining importance in mobile computing [1]. Firstly, smartphones are commonly used for various purposes in daily life. With rapid advancements in mobile computing, mobile devices equipped with multicore processors and GPUs are becoming ubiquitous. Powerful computation resources allow a wide range of x86_64 application binaries to be migrated to mobile devices transparently using DBT, e.g., multimedia and image applications. Mobile Edge Computing (MEC) augments mobile devices with powerful servers and provides mobile devices with opportunities to run computation-intensive applications by leveraging edge-based computational resources [2]. However, edge-based servers and smartphones typically adopt heterogeneous architecture, while code offloading which also relies on DBT is an ideal approach for leveraging high-performance resources on servers [3].

However, concerns about performance and translation efficiency constrain the use of DBT in mobile computing. There are two major factors that affect DBT performance: (1) translation and emulation overhead and (2) translated code quality. A DBT system translates one section of the guest binary code at a time to the host binary code and then executes it. The translation and emulation overhead of DBT and execution of the translated code together determine the overall performance of a DBT system. To achieve an effective trade-off between the two factors, a DBT system usually performs highly efficient optimizations: it cannot afford sophisticated and complicated optimizations, e.g., polyhedral optimization.

Ubiquitous multicore processors bring us opportunities to improve the performance of DBT. However, traditional DBT systems do not fully exploit the computational

resources provided by multicore processors, especially when translating sequential guest applications. Most studies have focused on leveraging multiple cores to reduce translation and emulation overhead. Approaches such as parallelizing loops in guest binaries and generating concurrent code assigned to multiple cores have been overlooked. One of the key benefits of this method is that DBT can thus achieve significant performance improvement when parallelizing hotspots of guest applications successfully, even outperforming native executing.

In this study, we develop LLPEMU, a DBT framework that can translate and parallelize affine loops in guest application binaries. LLPEMU consists of two components: a dynamic binary translator for nonhotspot translation and emulation and a static translator that extracts and parallelizes affine loops in guest binaries. LLPEMU uses an enhanced QEMU [4] as its dynamic binary translator and Polly [5] as the backend polyhedral optimizer of its static translator. However, instructions for the emulation mechanism in the translated code impede parallelization. We investigate these obstacles and present a feasible way to overcome them. With such a combined static-dynamic approach, we successfully perform polyhedral optimization while ensuring low runtime overhead. Our evaluation results show that LLPEMU outperforms QEMU significantly on PolyBench.

The main contributions of this study are as follows:

(1) We have developed a DBT framework with a combined static-dynamic design that performs polyhedral optimization on affine loops while incurring low runtime overhead

(2) We have proposed a loop-level DBT-specific optimization to provide opportunities to eliminate redundant loads/stores in the target code region

(3) We have investigated obstacles to parallelization that are created by the emulation mechanism of binary translation and presented a feasible method to overcome them

2. Background

As LLPEMU uses an enhanced QEMU as its dynamic binary translator, this section provides background information about QEMU. QEMU [4] is a popular open-source dynamic binary emulator which supports several ISAs such as x86, ARM, MIPS, and PowerPC. In this work, we use the process-level emulation of QEMU in LLPEMU.

QEMU is mainly composed of the two following stages: (1) Emulation: QEMU creates a virtual execution environment to emulate architecture states of virtual guest processors. Starting from parsing guest application executable file, binary code and data are loaded into QEMU memory space. While translating guest applications at the unit of basic block, architecture states of virtual processors stored in memory will be updated according to executing results. (2) Translation: QEMU facilitates quick emulation of multi-ISA by using Tiny Code Generator (TCG) [6]. Figure 1 draws an outline of TCG. The guest machine code on different ISAs is translated into intermediate representation by TCG. In this paper, we use TCG IR to refer to that. After highly efficient IR-level optimization, TCG IR is translated into the host machine code.

The main loop of QEMU is a translation-emulation loop. After QEMU initializes the virtual execution environment, translation starts from entry PC stored in architecture states of virtual processor, which is set during initialization based on information extracted from ELF. The guest code starting from entry PC is translated into TCG IR at the unit of basic block and then converted into the host machine code. The execution of the translated code will alter architecture states, and a new PC will be set. Then, the translation-execution loop restarts from the new PC.

All these works are done at runtime, leading to the conflict of high code quality and low overhead. In fact, QEMU just perform simple optimization liveness analysis and store forwarding on TCG IR. For short-running application, QEMU is an ideal choice. The design of TCG makes QEMU find the good balance between high-quality code and low overhead. Therefore, we choose to develop our binary translation framework based on QEMU.

But when running application with hotspots, especially multimedia applications with loop nests, QEMU does not perform well. Frequent control flow switching between execution of translated host code and QEMU dispatcher results in many load/store instructions to save program context. The cache hit rate has also been greatly reduced.

3. Architecture of LLPEMU

In this section, we first discuss the design issues that arise when developing a DBT framework that parallelizes loop nests in a sequential guest code by leveraging polyhedral optimization. Then, we describe LLPEMU in detail.

3.1. Design Issues. Our goal is to enable a DBT system to parallelize loop nests in the sequential guest code without incurring high runtime overhead. To this end, we focus on the following issues.

First, the runtime overhead due to optimization and parallelization of the target code should be as low as possible. However, analyzing and parallelizing binaries can be expensive. To parallelize the binaries, we must perform complex analysis to recover the CFG and extract the loops. A polyhedral optimizer introduces considerable overhead. Moreover, not all the loops can be parallelized. DBT can benefit from parallel execution only when the code is parallelized.

It is crucial to design the system architecture such that it can extract loops and perform polyhedral optimization without introducing high runtime overhead. On the basis of these considerations, we propose a combined static-dynamic DBT system. A dynamic binary translator is responsible for initializing the virtual execution environment and emulating guest CPUs; loop nests are extracted and parallelized statically ahead of time.

Although some studies have investigated trace formation based on dynamic profiling with acceptable overhead [7], these methods are not entirely suitable for extracting loop

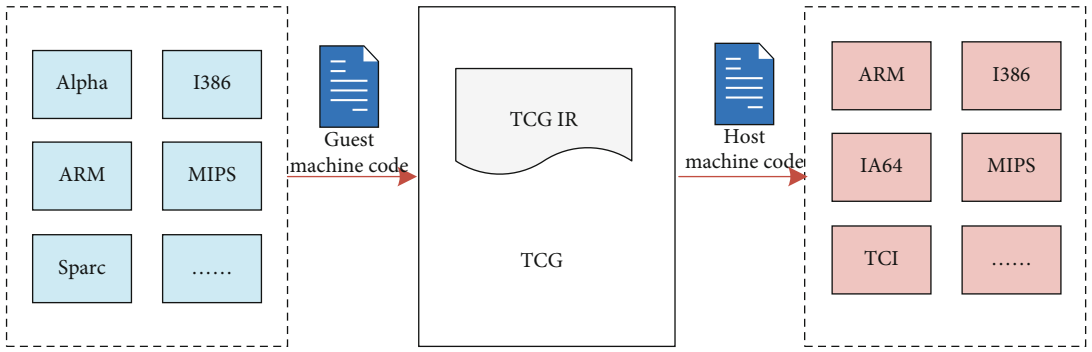

Figure 1: Outline of Tiny Code Generator (TCG).

nests. Trace selection and merging must be performed to format the loop nests. For better selections of traces, trace merging is usually performed on the basis of profiling, which involves high runtime overhead. Moreover, dynamic approaches cannot reduce the overhead of polyhedral optimization.

The second issue is how to parallelize loops extracted from guest binaries. Many polyhedral optimizers have been developed, such as PLUTO [8] and Polly [5]. However, nearly all of them focus on source-level optimization. In this study, we try to adapt these standard compiler tools to a DBT system to parallelize the guest machine code.

Adapting existing compiler methods designed for the source code to the DBT IR is a challenging task. Compiler tools perform analysis and transformations based on the loop structure and symbolic information. Heavy optimization and ISA-specific transformations during compilation make the loop structure and symbolic information unavailable. Thus, lifting the DBT IR up to the source code needs complicated decompilation.

On the basis of these considerations, we choose Polly as our parallelizer. Polly, which has been developed on the basis of LLVM infrastructure [9], analyzes and parallelizes loops at the IR level. Lifting TCG IR to LLVMIR does not require sophisticated decompilation. In addition, there are several general passes for analysis and optimization. We can use these passes to develop our transformation for TCG IR. Furthermore, LLVM can generate a machine code on various ISAs, which matches the retargetability of QEMU. Thus, LLPEMU can also support various host ISAs.

3.2. Overview of Architecture. The architecture and main components of LLPEMU are shown in Figure 2. Considering the adaptation of polyhedral optimization to binary translation with low overhead, LLPEMU consists of two translators.

3.2.1. Static Translator. In the static stage, loops are extracted from the guest application and then translated into the parallelized host machine code. Related information files are also generated to enable the dynamic binary translator to leverage the static analysis results and load the parallelized code. As an offline stage, any complicated analysis and optimization will not increase the runtime overhead. For a specific guest application, loops are extracted and optimized before the first run. However, every run can benefit from a high-quality host code.

3.2.2. Dynamic Translator. The dynamic binary translator is responsible for creating and maintaining the virtual execution environment of the guest application involving memory space mapping and state structure updating. The nonhotspot code is translated and executed in the dynamic stage. If the target PC is the entry of a target loop, the dynamic binary translator will switch the workflow from translation to the execution of a statically generated code.

Next, we describe the details of the two stages and discuss how the static and dynamic stages are combined.

3.3. Static-Dynamic Combination. The static-dynamic combination is central to the LLPEMU in order to achieve an effective trade-off between high code quality and low runtime overhead. It is crucial to determine how the static and dynamic stages can collaborate with each other. We must ensure that the parallelized code generated statically can be executed by DBT when translating the target code. The code switch must not impede the DBT emulation mechanism. In this section, we focus on the following issues.

3.3.1. Maintaining DBT Emulation. As translation and execution proceed, DBT alters both the data in the guest memory and the virtual processor state structure. When translating the target loops, DBT will redirect the control flow from the original translation-execution loop to the execution of the parallelized code generated in the static stage. Then, the translation-execution loop resumes with the end state of the parallelized code.

Therefore, the static translator should not only parallelize the sequential guest binaries but also ensure that the parallelized code it outputs maintains the DBT emulation. DBT can resume translating with the correct state only if the data in the guest memory and the state structure are altered as in the case of DBT.

Many studies have investigated lifting the machine code to the source code partially or to LLVM IR and then parallelizing the lifted code using compiler tools. Through these methods, the sequential guest machine code can be directly converted into the parallelized host code. However, such code cannot maintain the DBT emulation mechanism.

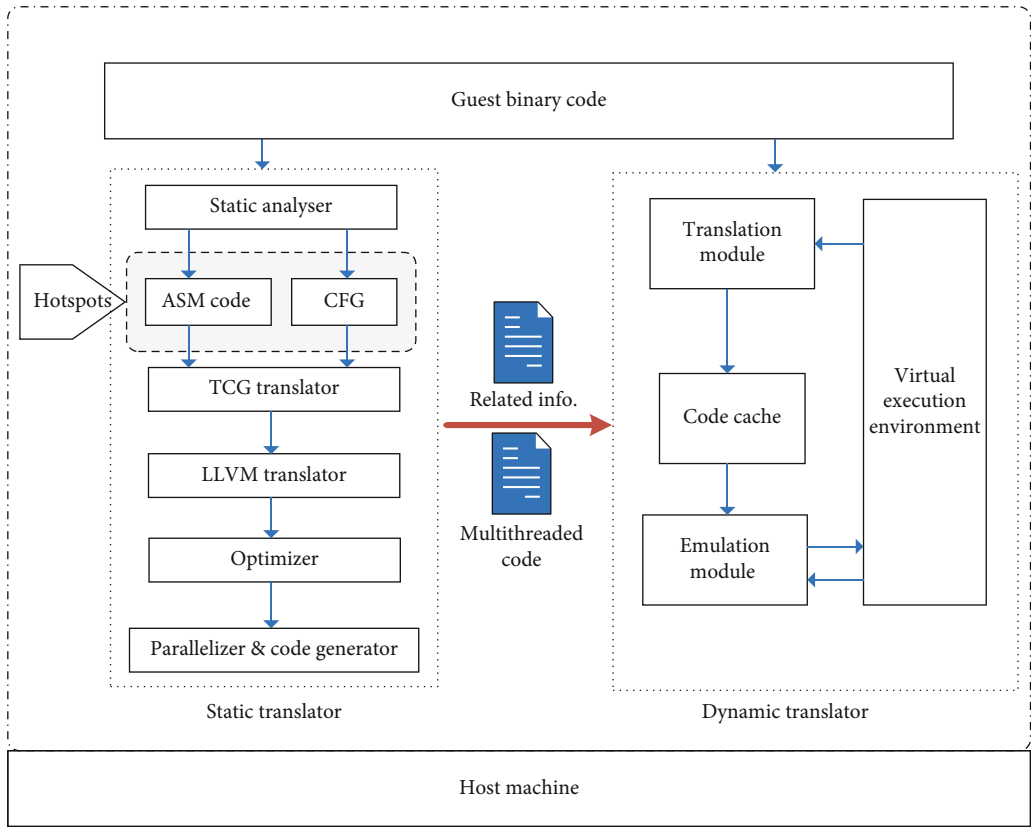

Figure 2: Overview of LLPEMU.

Next, we present our method that makes the static translator output code that is compliant with the DBT emulation mechanism.

First, a static TCG translator is developed to translate the guest machine code into TCG IR before conversion into LLVM IR. As described in Section 2, to emulate the virtual states, TCG will insert memory manipulation instructions to update the state structure in the memory. During execution of the translated code of a basic block, the state structure is altered according to the results of the instructions. Hence, we can exploit the TCG to generate a code with state structure manipulations.

Second, we perform memory space mapping. The base address of the guest application varies with each run. QEMU will choose an appropriate address instead of the address indicated by the ELF file. The target address of memory access to the guest application is computed on the basis of the base address. The actual address of the target memory access is calculated by adding the guest address in the virtual register to the base address. To overcome this obstacle, the base address is set as an argument of the package function; every memory access in packaged loops will be performed on the basis of this argument dynamically.

3.3.2. Interface Design. To leverage the parallelized code generated statically, the dynamic translator must know when and how to switch to the parallelized code. In the static stage, the code is compiled as a shared library using "-O3 -shared -PIC." Each target loop is packaged as a function with the required runtime component pointers as parameters, such as the pointer of the state structure and base address. The function name is a combination of the application name and entry PC of this loop; a specific target loop can be determined from the other loops by the two components.

The dynamic translator loads the shared library using POSIX API during initialization. *Dlopen()* and *dlsym()* are used to obtain a pointer to the function by a given name. Related information about the target loops is read from files to enable the dynamic translator to check whether the succeeding PC is the entry of a target loop. When the dynamic translator reaches the entry of a target loop, it triggers the redirection of the control flow to the execution of the optimized code. The advantage of using POSIX API is that we do not need to relink the optimized code.

3.4. Dynamic Binary Translator. The dynamic binary translator of LLPEMU is developed from QEMU, and additional functionality is added so that it can switch to a parallelized code generated statically when translating target loops. In the case of target loop nests, the control flow of the dynamic translator is switched to the execution of the parallelized code while QEMU translates and emulates the basic blocks one by one. As a result, the context switching overhead is reduced.

During the initialization of the dynamic translator, information about the parallelized loops, including the entry PC of the target loops and function name of the packaged loops, is read by parsing related information files. Based on this

information, the parallelized code is loaded and linked, and the function pointer to the packaged loops is also obtained.

The dynamic translator checks whether the entry PC of a basic block is also the entry of a parallelized loop to trigger the switch of the control flow. A simple approach is to perform a hash-table lookup when translating a new basic block. For a large-scale application with a few parallelized loops, the overhead incurred by a hash-table lookup is unacceptable.

To address this issue, we propose a low-overhead method. The key aspect of our method is avoiding a hash-table lookup by leveraging the code cache of the dynamic translator. As described in Section 2, the main loop of QEMU is a translation-emulation loop. Before translating a basic block, QEMU checks whether it has been added to the code cache. If it can be found, the translated code in the code cache will directly be executed. We already know the entry basic block of the target loop by parsing related information files. Then, the code cache of the entry basic block is generated during initialization. Code fragments in the code cache are not the translated code from the basic block but a prologue to redirect the execution from the code cache to the parallelized code. The parameters are also transferred to indicate the base address and pointer of the state structure.

This method makes it possible to check for the entry of the target loops without performing a hash-table lookup when translating every basic block. The decision of triggering the switch of the control flow is hidden when finding the code cache, which QEMU originally needs to do. Nearly no runtime overhead is incurred by this method, except for the initialization of the target code cache.

3.5. Static Translator. Static translation starts from extracting loops from guest binaries. From the assembly code, the control flow graph (CFG) is reconstructed [10]. Code regions where the CFG cannot be reconstructed statically will be discarded. For there are always back edges in CFG of loops, it is easy to extract loops from guest binaries. Then, we attempt to extend these loops to the outermost to find loop nests, considering that we can benefit more from larger loop nests. All found loop nests are marked as a candidate for further optimization.

An initial estimate is made to check if we can benefit from parallelization of one loop nest. Here, we simply use loop counts as the threshold. We insert runtime check instructions to roll back to the bare sequential version if no benefits can be gained.

In the following, we show details about how we translate and parallelize these candidate loop nests.

3.5.1. Translating Machine Code to LLVM IR. The static translator translates the binary code of the target loops to the LLVM IR [9] for further optimization. To maintain the DBT emulation, as described in Section 3.3, the binary code is translated to TCG IR before conversion to LLVMIR. The translation is implemented by the TCG translator and the LLVM translator seamlessly.

The TCG translator is developed from the TCG of QEMU. We split the TCG from QEMU and extend it to form a static one. The TCG translator translates the machine code into the TCG IR at the granularity of a basic block. A basic block is defined as a single-entry single-exit region of a code with a control flow instruction at the end.

However, the original TCG cannot check whether the next instruction is the entry of another basic block and only finishes formulating a basic block when it meets a control flow instruction. As a result, basic blocks generated from loop nests by the TCG usually overlap, leading to complex control flows and redundant code.

To solve this problem, we insert a jump instruction to the succeeding instruction as the end of the basic block when we find that the next instruction is the entry of another basic block based on CFG.

Next, the TCG IR of basic blocks is taken by the LLVM translator as input. First, the TCG IR is seamlessly translated into LLVM IR by one-to-one mapping. Then, the basic blocks of one target loop are packaged as a function in LLVM IR, and attributes are added to provide prior information about DBT. Helper functions in TCG designed to emulate flags and soft-float computation are inlined. Thus, additional opportunities are provided to carry out loop-level optimization and make it easier for Polly to analyze these loop functions.

3.5.2. Optimization and Parallelization. The LLVM translator outputs an equivalent LLVM IR of loop nests with emulation instructions. However, the parallelizer cannot automatically parallelize target loops by directly taking lifted IR generated by the LLVM translator. In fact, as the parallelizer, Polly always fails by directly taking lifted IR as input. We also find that even for nested loops that can be parallelized by Polly in the form of a source code, IR translated from its sequential machine code cannot be optimized by Polly.

Meanwhile, the code organized at the loop level instead of the basic-block level provides us with opportunities to perform sophisticated optimization such as dataflow analysis. As QEMU performs only simple optimization within a basic block, there is still redundant code to be eliminated. Although there are many aggressive optimization passes, the LLVM optimizer cannot handle these issues completely owing to a lack of prior knowledge of the DBT emulation mechanism.

To overcome these issues, we present a feasible method for optimizing the unoptimized IR at the loop level and provide opportunities for polyhedral parallelism. The optimizer is developed on the basis of prior knowledge of DBT and implemented as LLVM passes to bridge the gap between the translator and Polly. Then, IR optimized by the optimizer will be taken into the parallelizer. Finally, the code with polyhedral optimization is output by Polly and compiled into an executable code as a shared library.

4. Towards Polyhedral Optimization

Since we try to adapt Polly which is designed for the source code to binaries, we use the static translator to transform binaries into LLVM IR. As described in Section 3.3, the TCG translator and the LLVM translator enable this transformation. However, directly taking the unoptimized IR

output by the LLVM translator as input, Polly cannot perform automatic parallelization on the binaries successfully. Therefore, to bridge the gap between the translated IR and the IR that Polly can handle, the optimizer has been inserted between the parallelizer and the LLVM translator in the static translator. Here, we present our method implemented in the optimizer for optimizing and transforming unoptimized IR into a Polly-friendly version.

According to our previous evaluation, we find that there are two major factors that impede Polly: (1) DBT emulation instructions are inserted into IR during translation from the machine code to TCG IR. As LLVM IR are converted from TCG IR by one-to-one mapping, these emulation instructions are not optimized. Complicated memory manipulation for DBT emulation makes IR complicated for Polly; (2) optimization during compilation makes the loop structure and symbolic information unavailable, and x86 registers are used for calculation and address computation. From these complex instructions, Polly cannot obtain the information it needs to model the target loop. In the following, we will elaborate on the solution to these two problems.

4.1. Loop-Level DBT-Specific Optimizations. In this subsection, we focus on redundant code elimination on translated IR. Only simple optimization like liveness analysis and store forwarding is carried out within basic blocks by the TCG translator. The LLVM translator transforms TCG IR to LLVM IR in a way of one-to-one mapping and integrates the basic block to the nested loop region with CFG completely known. Organized as loop nests instead of basic blocks, the code with high-level structure information makes it possible to carry out sophisticated optimization like data flow analysis and other aggressive optimizations.

LLVM passes such as common subexpression elimination and instruction combination can be used to optimize some rather simple redundant codes. But due to the lack of prior knowledge of the DBT mechanism, redundant memory manipulations related to the DBT emulation mechanism are overlooked. We all know that redundant memory access will incur considerable overhead.

In the following, we present our method to eliminate redundant memory manipulations caused by DBT emulation at the loop level.

4.1.1. Optimizing Virtual Register Emulation. The dynamic translator allocates the state structure in the global memory for each virtual processor to model its architecture state. Data in virtual registers are read and written by pointers into the state structure. When translating the x86 machine code into TCG IR at the granularity of a basic block, several load/store instructions are generated to alter the architecture state. However, nearly all such memory access to the state structure is redundant when the basic blocks are organized as a loop.

A typical example is given in Figure 3. We can see that the value in virtual register *%rax* is loaded immediately after the store instruction without any intervening stores to this state. Here, we can use the value *%rcx.0* to replace all use of the loaded value and thus forward the *store* to the *load*. After this

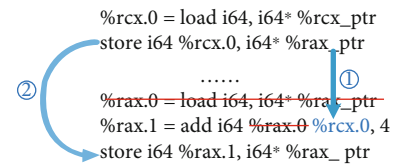

Figure 3: An example of substitution forwarding.

forward substitution, the store instructions are also exposed to be optimized by store sinking.

Then, we carry out such optimization at the loop level. With CFG completely known, such forward substitution and store sinking are performed across the loop nests. All redundant store and load instructions to update the virtual state structure within the loop bodies are eliminated. And only in the exit basic block, the store instructions are inserted to update the state structure according to the execution results of the last iteration.

4.1.2. Stack and Frame Manipulations. The stack and frame structure in x86 architecture is used to store the intermediate result of computation and program context. Memory accesses into stack and frame are performed by using frame pointer register *%rbp* and stack pointer register *%rsp*. As shown in Figure 4, the memory access into the stack is converted to a sequence of memory operations in LLVM IR. Such stack manipulation instructions cannot be handled by general loop passes.

If comparing stack memory cells in guest space to virtual states in QEMU space, we will find that memory manipulations on these memory cells behave the same. States in state structure can be addressed by pointers and constant offsets, while stack memory cells are addressed by *%rsp*/*%rbp* and offsets. To simplify complicated memory manipulations for emulating x86 stack and frame, we apply forward substitution and store sinking to these stack memory cells the same as to virtual states.

4.2. Tailoring IR for Polyhedral Optimization. Polly is a polyhedral optimization tool based on LLVM infrastructure. It is implemented as a set of LLVM passes, to perform analysis and transformations on IR. As a compiler tool, Polly is designed to optimize IR generated by Clang, and some passes in Polly rely on results of general pass such as the loop pass and SCEV pass.

Polly first formulates the polyhedral model of loops, starting with the detection of static control parts (SCoPs) [5] in loops. Only loop nest parts detected as SCoPs will be optimized by Polly. When a region cannot be detected as SCoPs, Polly will terminate the analysis on this region immediately.

Successful analysis of induction variables and memory access is essential to Polly. Polly obtains information about induction variables from the SCEV pass and uses the ISL library [11] to extract information about the stride and loop-trip count. A GEP instruction is a special instruction in LLVM IR that is used to calculate the target address of structural memory access. By taking the structure definition

x86 assembly	Unoptimized IR
L : … add rax , -0x8 (%rbp) … jl L …	L : … %rbp.0 = load i64, i64* %rbp_ptr %33 = add i64 %rbp.0, -8 %34 = inttoptr i64 %33 to i64* %35 = load i64, i64* %34 %36 = add i64 %rax.0, %35 store i64 36%, i64* %34 … br label L …

Figure 4: An example of stack manipulations.

of the array, pointer to array, index of target element, and type information as input, a GEP instruction returns the target address of array access. Polly obtains the structure information of array access from the GEP instructions.

From the above-mentioned description, we can determine that without heavy and architecture-specific optimization, loop structure information and symbolic information remain in IR generated by Clang. Passes on which Polly is dependent can obtain the loop structure information. However, translated IR is far more complicated than such unoptimized IR; hence, Polly always fails by directly taking the translated IR as input.

Next, we show these structure gaps and present our approach to seamlessly transform translated IR into a form suitable for Polly.

4.2.1. Memory Access Pattern. We already know that Polly obtains memory access patterns by analyzing GEP instructions, which involves the structure information of array access. However, address computation of array access in translated IR, by taking the values in virtual registers as operands, is performed similarly to that in the case of x86 architecture. As a result, Polly cannot reconstruct memory access patterns of translated IR.

Polyhedral optimization mainly targets affine loop nests, which requires the array index of array access of an affine function of induction variables. Symbolic descriptions are used to represent the array index of the affine memory address. For example, if i is the induction variable of array A, then $A[i]$ represents the ith element of array A and memory references $A[i]$ and $A[2i+1]$ are affine, whereas $A[i/4]$ is not.

To convert IR into a form suitable for Polly, it is crucial to know how the translated IR is organized to compute the address of array access in affine loops. On x86-64 architectures, the target address of memory access is usually computed as

$$\text{Target\_addr} = \text{Addr\_base} + s * \text{Index} + \text{Offset}, \quad (1)$$

where Addrbase and Index are values stored in registers, while s and Offset are immediate values. Such address computation can be performed within one instruction by the underlying x86 architectures.

There are structure gaps between LLVM IR and x86 assembly code. LLVM IR is of the SSA form, and every value is attached with explicit type information.

An example is shown in Figure 5. The x86 address computation operation is converted to multiple instructions in LLVM IR with the pattern of "calculation-inttoptr-bitcast." "Calculation" refers to operations that compute the target address by taking integers in virtual registers as input. Then, an inttoptr instruction is used to convert integer to pointer type in LLVM, taken by the load/store instruction as input. These structure gaps make it hard for Polly to model target loops.

4.2.2. Reconstructing Loop Structure. In the following, we present our method to reconstruct target loops from the translated IR. The key strategy is to recover the loop induction variables and memory access symbolic description. Based on recovered loop structure information, the translated IR is then transformed into an equivalent version with GEP instructions, which is suitable for polyhedral optimization on Polly.

To obtain symbolic description of memory access, we start from the PHI node in LLVM IR. With an initial value, the variable defined by the PHI node is modified in each iteration. We attempt to reconstruct the induction variable from these PHI nodes. The SCEV pass [12] is used to obtain the symbolic description of induction variables.

In particular, a normalized loop counter is introduced for each loop when there is no normalized one. Starting from zero, a normalized loop counter is incremented by one in every loop iteration. When the stride of a variable defined by the PHI node is constant, we can obtain its symbolic expression described by the normalized loop counter.

Then, we try to reconstruct the symbolic expression of memory access. By analyzing the pattern of calculation-inttoptr instructions, memory access is marked as a candidate for reconstruction. A symbolic expression of memory access is rebuilt using definitions involved in the address computation. This process is performed recursively by following def-use chains, until it reaches a known induction variable, an argument, or a loop-invariant variable.

When all the variables participating in the address computation in a loop, excluding loop-invariant variables, can

x86 assembly	Unpotimized IR
Addsd (%rcx, %rdx,1), %xmm0	%21 = add i64 %rcx, %rdx %22 = inttoptr i64 %21 to i64* %23 = bitcast i64* %22 to double* %24 = load double, double* %23

Figure 5: Difference of address computation.

be described by the affine functions of loop indices, this loop can be marked as a candidate.

For loop nests, the outer loop induction variable is usually the initial value of an inner loop induction variable. Such expression substitution and resolution are performed recursively from the innermost to the outermost one. Finally, we can always obtain the symbolic description of affine access in nested loops, expressed as

$$\text{Target\_addr} = \text{Addr\_base} + \sum_{k=1}^{n} \text{index}_k * \text{stride}_k, \quad (2)$$

where index_k is the induction variable of the kth loop dimension and stride_k is the loop stride. Addrbase is the starting address of an array.

When we obtain all the symbolic description of induction variables in nested loops, we have all the information of array accesses. However, to transform these instructions into GEPs, we need data type information of array elements. Note that in x86, variables in registers are not attached with explicit type information. Variables are stored in general-purpose registers. Type information of nearly all of them can be recovered only through the way it is used.

When translating an x86 instruction, TCG will specify a general type for a variable based on its size. When the general type is conflicting with operation input, TCG then convert it to the correct type. After computation, results are converted back to the original data type and wrote back to virtual registers or memory cells. Many *trunc* and *bitcast* instructions are inserted into IR to facilitate these data type conversions.

To solve this problem, we recognize the data type participating in calculation based on operations on data and unit size of the array element. To do this, we start from *inttoptr* and *bitcast* instructions to obtain correct data type information in calculation. Then, the correct data type is propagated by following def-use chains until we meet load instructions. As values are loaded from virtual registers or memory cells, we consider that we find the source of them. After data type resolution, the related instructions are modified to generate GEP instructions.

4.2.3. Separating Emulation Instructions. If a target loop is successfully reconstructed by our optimizer, Polly can detect SCoPs in it successfully and formulate the polyhedral representation. Then, dependence analysis is carried out to determine whether a loop is parallel. Even for loops which can be parallelized in the form of the source code, IR lifted from its machine code always fails to be marked as parallel by Polly. In this section, we attempt to address this issue.

We found that virtual register updating instructions introduce loop-carried dependencies to loop nests. As described in Subsection 4.1, expression forward substitution and store sinking are performed to eliminate redundant load/store instructions for emulating the virtual states. Only in the exit blocks of loop nests, virtual register states are updated according to intermediate results of instructions in the last iteration. However, for single-thread guest application, there is only one virtual CPU emulated by QEMU. These instructions with memory accesses to the virtual states introduce loop-carried dependencies to target loops because virtual states are global variables existing in QEMU space.

Since only intermediate results of instructions in the last iteration are used to alter virtual states, we separate the last iteration from the others to overcome this problem. Here, we assume that the loop-trip counts are constants, so it is easy to determine which is the last iteration. The target loop is split up into two regions: instructions in the last iteration and the kernel region with the other iterations. As a result, loop-carried dependencies caused by emulation instructions are removed from the kernel region. Then, the kernel region is fed to Polly to be parallelized.

5. Evaluation

5.1. Experimental Setup. In this section, we present the performance evaluation of LLPEMU. We conducted the experiments with the emulation of PolyBench4.2 benchmark [13], which consists of kernels in multimedia applications. Here, we choose 6 programs of them which are optimized well by Polly. We evaluated the performance of LLPEMU with x86_64-to-ARM emulation. With an ARM board as the host machine, LLPEMU takes x86_64 guest binaries (compiled by Clang -O2) as input and generates parallelized ARM binaries. The dynamic translator of LLPEMU is developed from QEMU version 5.1, and we use Polly version 6.0 as the polyhedral optimizer. The detailed hardware and software configurations are listed in Table 1.

5.2. Speedup of Loop-Level Optimizations. Firstly, we check the results of LLPEMU running test programs and confirm that the results generated by LLPEMU are the same as those from QEMU. The loops in kernels of test programs are successfully detected and replaced by the optimized code. Then, the performance of LLPEMU is compared to that of QEMU. We use the running time of QEMU translating kernel

Table 1: Hardware and software configurations.

Processor	ARMv8 (big.LITTLE)
Frequency	Up to 2.0 GHz
Cores	2 cortex-A72 cores+4 cortex-A53 cores
OS	Ubuntu 18.04
Linux kernel	4.4.179
Memory	2 GB

functions as the baseline. The dynamic running time of LLPEMU translating kernel functions is measured, and the speedup achieved by LLPEMU can be computed as

$$\text{Speedup} = \frac{\text{Time}_{\text{QEMU}}}{\text{Time}_{\text{LLPEMU}}}. \quad (3)$$

The performance of a simple version of LLPEMU only with loop-level optimizations and the full version with polyhedral optimization is shown in Figure 6. The y-axis is speedup ratios achieved by LLPEMU against QEMU. The results show that LLPEMU (simple) speeds up all six program executions and reaches an average of 5.0x. We also observe that LLPEMU (simple) achieves speedup ratios from 3.3x up to 9.0x.

The reason why loop-level optimizations can lead to such improvement mainly comes from two aspects. First, high-level structure information allows us to perform sophisticated optimizations, leading to high code quality. Second, taking the loop region as the translating unit reduces context switch overhead. When QEMU redirects its control to the execution of the translated code, the program context must be saved. And after execution, the context will be restored for control to come back to QEMU. These switches lead to lots of memory accesses which incur significant overhead.

Our second set of experiments is aimed at evaluating whether the method we proposed is feasible enough to transform unoptimized IR into the Polly-friendly version. Different from the first set of experiments, IR generated by the LLVM translator is optimized by the optimizer before taken into the parallelizer. Speedup times are calculated, also with the execution time of QEMU as the baseline.

From the results, we observe that the translated IR (except *syrk*) are parallelized by Polly successfully, and LLPEMU (full) gains considerable speedup times from the simple version with only loop-level optimization. Five of the benchmark programs achieve more than 6x speedup compared with QEMU, and four of them are more than 10x. The results demonstrate that our method successfully reconstructs loops in translated IR and transforms them into a form suitable for Polly. SCoPs in kernels have been detected and provide information to enable Polly to check whether they are parallel. It is clear that the method proposed to overcome obstacles preventing parallelization is feasible.

It is seen that LLPEMU (full) fails to perform parallelization transformation on *syrk* and achieves similar performance to its simple version. This comes from parametric loop bounds used in an inner loop of the *syrk* kernel. Our method cannot handle parametric loop bounds, and loops with it will not be marked as a candidate. Parametric loop bounds are usually stored in registers which require more complex analysis and symbolic derivation to recover loop bounds from binaries. Kotha et al. [14] have proposed a method to extract parametric loop bounds from registers in a pattern-matching way. Because Polly can analyze parametrized loops, we believe that this problem can be handled by LLPEMU integrated with more sophisticated analysis and transformations.

5.3. Quality of Parallelized Code. This evaluation is aimed at examining the quality of the parallelized code generated by LLPEMU and measuring how many opportunities our method provides for Polly to perform polyhedral optimizations. Figure 7 presents the performance of the native ARM code, kernel code generated by LLPEMU in 6 threads, and native ARM code parallelized by Polly. Here, the native code is compiled using "Clang -O2." "Polly-6" represents for the native code parallelized by Polly in 6 threads.

From the results, we observe that the kernel code generated by LLPEMU is even better than the native code with an average 3.09x speedup. This observation supports the importance of leveraging polyhedral optimization in DBT. Because kernels in benchmark programs are all loop nests with simple computation, parallelization results in a large portion of performance improvement. Thus, the performance of code generated by LLPEMU differs from that of "Polly-6" slightly.

Next, we move to the comparison of LLPEMU and "Polly-6." From the results, we observe that LLPEMU on *gemver* is only about 40% of "Polly-6." The reason that causes slowdown is that our method fails to eliminate all loop-carried dependencies. We found that loop-level optimizations eliminate most of those dependencies, but it is still required to perform more advanced optimizations to eliminate them all. If we eliminate such dependencies, we could still achieve performance improvement like the other four programs.

It is also seen that LLPEMU is always slower than "Polly-6" with an average of 90% excluding *gemver*. The reason for this phenomenon is that there are many additional instructions in kernels generated by LLPEMU. As described previously, assumptions are made and runtime validation needs to be done in kernels to make sure those assumptions are right. If not, kernels will roll back to a sequential version instead of a parallelized one. Besides, emulation instructions and manipulations to package the kernel function to a Polly-friendly version also reduce the speed.

6. Related Work

The novelty of this study is the proposal of a feasible way to improve the performance of a dynamic binary translation system by leveraging polyhedral optimization at the loop level. Beyond automatic parallelization, this study provides opportunities for further optimization, such as automatic vectorization [15] and hotspot offloading [16], to platforms with more powerful computation resources.

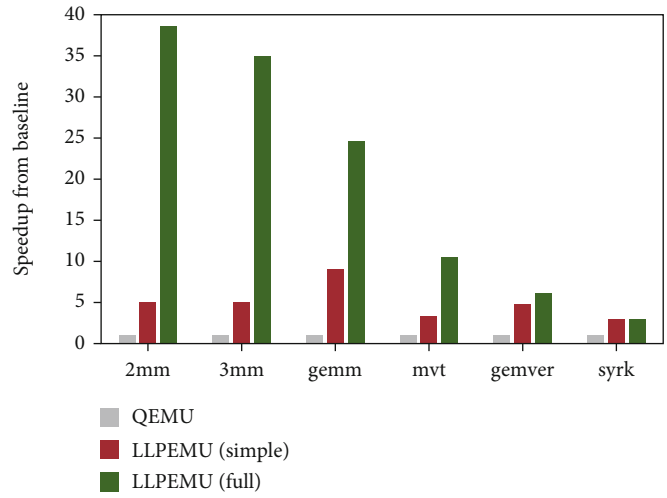

Figure 6: Performance comparisons of LLPEMU and QEMU.

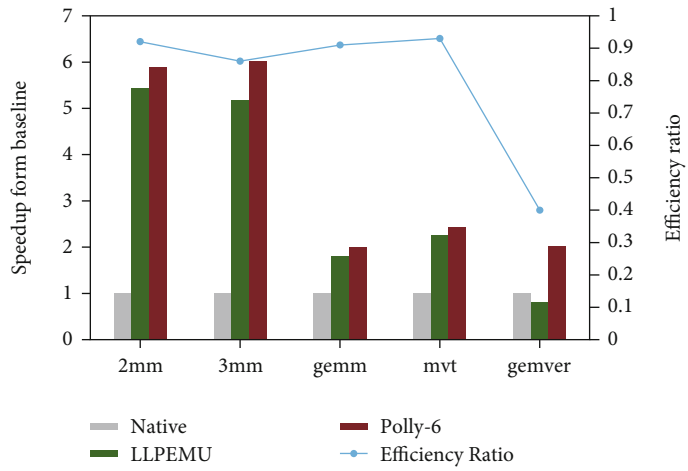

Figure 7: Quality of parallelized code.

By leveraging DBT for various purposes, such as cross-ISA emulation, transparent optimization [17], and profiling [18], many optimization approaches have been proposed. This paper is aimed at achieving better performance when migrating multimedia applications across ISAs using DBT. To this end, we leverage polyhedral optimization to generate the concurrent code that exploits abundant multicore resources.

6.1. Binary Translation. To the best of our knowledge, the approach that is the most closely related to LLPEMU is HQEMU [7]. HQEMU is a trace-based dynamic binary translator that is also built on QEMU and uses LLVM as the backend optimizer. Small sections of the code are inserted at the beginning of each translated code region to obtain profiling information. Based on profiling information, trace detection and merging are performed. Then, the traces are optimized and translated into the host machine code by optimizers on different processor cores. This approach involves profiling and trace optimizing at runtime, and it does not carry out polyhedral optimization to generate the concurrent code. In its DBT system, optimizers are attached to different processors to reduce the runtime overhead. By contrast, LLPEMU detects and parallelizes loops statically. Thus, with high-level information of the target loops, sophisticated optimization can be performed while incurring low runtime overhead. Furthermore, instead of using multiple cores to reduce the optimization overhead, LLPEMU generates a concurrent code to directly improve the execution performance.

Some other multithread DBT systems have also been proposed. COREMU [19] emulates multiple cores in the full-system model. Its system is parallelized by creating multiple QEMU emulators and assigning them to multiple threads. In a different way, Ding and Chang [20] parallelize QEMU internally and propose a method to arrange critical sections carefully. Their goal is to improve the performance of emulating multithread applications by parallelizing DBT systems, and they did not perform sophisticated optimization on the translated code. Their DBT systems cannot benefit as much when translating sequential applications, whereas our system

is mainly aimed at parallelizing loops in sequential applications.

6.2. Binary Parallelization. There has been some prior work in binary parallelization. Sato et al. [21] proposed a dynamic code parallelization system (ExanaDBT) that also applies Polly to LLVM IR recovered from the binary code. However, their approach cannot work with DBT, and they assume that target loops do not contain access to global variables. By contrast, virtual states in binary translation are stored as global variables, and most of our work is related to these global variables. Pradelle et al. [22] partly lift binaries to C code and fed them to a polyhedral compiler. Kotha et al. [14] proposed a static binary parallelizer, which uses a dependence vector to determine whether a given loop should be parallelized. However, all these approaches are aimed at transparently parallelizing binaries within the same ISA. Instead, our system translates and parallelizes loops in guest binaries into a cross-ISA concurrent code and supports various architectures, as it uses retargetable tools QEMU and LLVM.

7. Conclusions and Future Work

In this paper, we presented LLPEMU, a dynamic binary translation framework that automatically parallelizes the guest binary code. LLPEMU translates and parallelizes loop nests in the guest binary code statically and switches to the parallelized code at runtime. The static-dynamic combined design allows LLPEMU to perform analysis and polyhedral optimization with low runtime overhead. Further, we investigated factors that impede parallelization of translated IR by the polyhedral optimizer of LLPEMU and proposed a feasible method to overcome these obstacles.

We have evaluated the performance of LLPEMU on the PolyBench benchmark. The results show that LLPEMU successfully performed translation and parallelization on loop nests in x86_64 binaries and achieved a considerable performance improvement over QEMU and even over the native sequential code in some cases.

In the future, this work must be extended and improved from the following aspects: (1) performing more powerful analysis and more aggressive optimizations to remove data dependencies in reconstructed loop nests and (2) extending the ability of our method to handle binaries compiled with more aggressive optimizations.

Data Availability

All the data is available online.

Conflicts of Interest

The authors declare that there are no conflicts of interest regarding the publication of this paper.

Acknowledgments

This research is supported in part by the National Natural Science Foundation of China under Grant 61472447, Grant 61802433, and Grant 61802435.

References

[1] E. R. Altman, D. Kaeli, and Y. Sheffer, "Welcome to the opportunities of binary translation," *Computer*, vol. 33, no. 3, pp. 40–45, 2000.

[2] Y. Mao, C. You, J. Zhang, K. Huang, and K. B. Letaief, "A survey on mobile edge computing: the communication perspective," *IEEE Communications Surveys & Tutorials*, vol. 19, no. 4, pp. 2322–2358, 2017.

[3] J. Shuja, A. Gani, M. H. Rehman et al., "Towards native code offloading based MCC frameworks for multimedia applications: a survey," *Journal of Network and Computer Applications*, vol. 75, pp. 335–354, 2016.

[4] F. Bellard, "QEMU, a fast and portable dynamic translator," in *USENIX Annual Technical Conference, FREENIX Track*, vol. 41, p. 46, Berkeley, CA, USA, 2005.

[5] T. Grosser, A. Groesslinger, and C. Lengauer, "Polly—performing polyhedral optimizations on a low-level intermediate representation," *Parallel Processing Letters*, vol. 22, no. 4, article 1250010, 2012.

[6] Tiny code generatorhttp://wiki.qemu.org/Documentation/TCG.

[7] D. Y. Hong, C. Hsu, P. Yew et al., "HQEMU: a multi-threaded and retargetable dynamic binary translator on multicores," in *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, pp. 104–113, San Jose, CA, USA, 2012.

[8] A. Mignone, G. Bodo, S. Massaglia et al., "PLUTO: a numerical code for computational astrophysics," *The Astrophysical Journal Supplement Series*, vol. 170, no. 1, pp. 228–242, 2007.

[9] C. Lattner and V. Adve, "LLVM: a compilation framework for lifelong program analysis & transformation," in *International Symposium on Code Generation and Optimization, CGO 2004*, pp. 75–86, Palo Alto, CA, USA, 2004.

[10] Y. Sato, Y. Inoguchi, and T. Nakamura, "Identifying program loop nesting structures during execution of machine code," *IEICE Transactions on Information and Systems*, vol. E97.D, no. 9, pp. 2371–2385, 2014.

[11] S. Verdoolaege, "isl: an integer set library for the polyhedral model," in *International Congress on Mathematical Software*, pp. 299–302, Springer, Berlin, Heidelberg, 2010.

[12] R. V. Engelen, "Efficient symbolic analysis for optimizing compilers," in *International Conference on Compiler Construction, (CC2001)*, pp. 118–132, Springer, Berlin, Heidelberg, 2001.

[13] PolyBench PolyBench, 2017, https://sourceforge.net/projects/polybench/.

[14] A. Kotha, K. Anand, M. Smithson, G. Yellareddy, and R. Barua, "Automatic parallelization in a binary rewriter," in *2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 547–557, Atlanta, GA, USA, December 2010.

[15] N. Hallou, E. Rohou, and P. Clauss, "Runtime vectorization transformations of binary code," *International Journal of Parallel Programming*, vol. 45, no. 6, pp. 1536–1565, 2017.

[16] M. Damschen, H. Heinrich, G. Vaz, and C. Plessl, "Transparent offloading of computational hotspots from binary code to Xeon Phi," in *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2015*, pp. 1078–1083, Grenoble, France, 2015.

[17] M. A. Watkins, T. Nowatzki, and A. Carno, "Software transparent dynamic binary translation for coarse-grain

reconfigurable architectures," in *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 138–150, Barcelona, Spain, March 2016.

[18] I. Böhm, B. Franke, and N. Topham, "Cycle-accurate performance modelling in an ultra-fast just-in-time dynamic binary translation instruction set simulator," in *2010 International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation*, pp. 1–10, Samos, Greece, July 2010.

[19] Z. Wang, R. Liu, Y. Chen et al., "COREMU: a scalable and portable parallel full-system emulator," in *Proceedings of the 16th ACM symposium on Principles and practice of parallel programming - PPoPP '11*, pp. 213–222, San Antonio, TX, USA, 2011.

[20] J. H. Ding and P. C. Chang, "PQEMU: a parallel system emulator based on QEMU," in *Proceedings of the 2011 IEEE 17th International Conference on Parallel and Distributed Systems, IEEE Computer Society*, vol. 10, Washington, 2011.

[21] Y. Sato, T. Yuki, and T. Endo, "ExanaDBT: a dynamic compilation system for transparent polyhedral optimizations at runtime," in *Proceedings of the Computing Frontiers Conference*, pp. 191–200, Siena, Italy, May 2017.

[22] B. Pradelle, A. Ketterlin, and P. Clauss, "Polyhedral parallelization of binary code," *ACM Transactions on Architecture and Code Optimization*, vol. 8, no. 4, pp. 1–21, 2012.