High-Level Optimizations for Newly Created and Existing Microprocessor Architectures

Drozdov Alexander Yu., Vladislavlev Victor E.
Department of Radio Engineering and Cybernetics,
MIPT, Moscow, Russia
alexander.y.drozdov@gmail.com
victor.vladislavlev@gmail.com

Novikov Sergey V., Kochetkov Eugeniy L.
SMWare Company
Moscow, Russia
serg.v.novikov@gmail.com
eugene.kochetkov@gmail.com

***Abstract*— This work is devoted to the description of a compiler designed based on LLVM. The LLVM infrastructure has an integrated UTL library containing implementation of analysis and optimization algorithms – auto-parallelizing and vectorization in particular. Using benchmarks from the SPEC/CPU2006 test suite, the obtained solution is compared with a Linaro GCC compiler on an ARMv8 architecture multi-core processor and with GCC and Clang on a dual-core MIPS processor. The transformations implemented in the UTL allowed achieving a significant performance gain. Besides, the work describes experience of adding back-end to LLVM for a new DSP architecture Lynx developed by the Milandr RC.**

Keywords*— *multi-core; optimizations; optimizing compiler; auto-parallelizing; vectorizing; LLVM; DSP-processor

I. INTRODUCTION

High performance computing is primarily associated with powerful servers, e.g. those on x86 architecture processors. To achieve maximum performance, detailed elaboration and configuration extends to not only computers but also memories, interconnection and, of course, compilers. The icc compiler developed by Intel Corporation can be regarded as reference for the x86 architecture. Sun's compiler for the same architecture is highly competitive with the Intel's one. Such widespread open source compilers as GCC [1] or Clang created on LLVM base [2] are notably inferior to the said proprietary solutions.

There are no such high-grade compilers for the architectures which have never participated in the race for top performance. The ARM architecture, so popular now, is supported by the Linaro community; its purpose is improving open source solutions, including the GCC compiler for this architecture [3]. However, in the case of open sources, it is not possible to add a technique like, for example, auto-parallelizing, present in Intel or Sun solutions. There is a number of third-party libraries, but they are not capable to demonstrate decent results for real tasks, e.g. the SPEC benchmarks [8].

Meanwhile, the need for such modified solutions for this architecture is expanding. Not so long ago, there was a discussed case of Chinese search engine Baidu which moved its computing infrastructure from x86 to ARM. The company claimed that the transition allowed reducing its total costs for computing system maintenance by one quarter. Today such transitions are frequent. The Russian company Rikor has recently developed a data warehouse based on ARM servers.

The industry is preparing for ARM architecture to become a full-fledged player on the market of servers. Qualcomm has announced they are going to release a 24-core processor for servers based on 64-bit architecture ARMv8. Cavium, Broadcom, Applied Micro, AMD and others declared developing ARM processors for servers before Qualcomm.

Imagination Technologies Group works actively on new generations of MIPS architecture and plans to become a serious rival for both x86 and ARM. The company Baikal Electronics was the first to produce a processor with new MIPS cores Warrior P5600. However, MIPS has no community like Linaro – that is why open source solutions (first of all, compilers) provide only basic support for this architecture. It leads to certain performance issues, their examples are given below. Even if open source solutions are improved for particular architectures (like in situation with Linaro), all the changes are limited to supporting architecture-specific parts. We offer a different approach. The Universal Translation Library (UTL) [4] developed by the authors contains most advanced algorithms of code analysis and optimization [5], as well as a mechanism for integrating these algorithms into arbitrary translation chain – into GCC and LLVM in particular [7]. This work describes several classes of the high-level transformations which allowed achieving functionality comparable with best proprietary solutions:

- Memory access optimization,
- Auto-parallelizing,
- Vectorization.

To evaluate the quality of the implemented algorithms, we integrated UTL with LLVM compiler Clang. The obtained solution was called SmartCC, abbreviated as SMCC. This review tells about the results of comparing our solution with the best available Linaro GCC compilers and the LLVM compiler on multi-core platforms with ARM and MIPS architectures using SPEC/CPU2006 benchmarks.

There are new architectures that also need effective development tools. In this paper we share our experience of porting LLVM to the Lynx, a DSP processor, developed by the Milandr RC [10]. Several features of this processor architecture did not allow applying the available

This work is supported by the Ministry of Education and Science of Russian Federation under contract No02.G25.31.0061 12/02/2013 (Government Regulation No 218 from 09/04/2010).

978-1-4673-8482-7/15 $31.00 © 2015 IEEE
DOI 10.1109/EnT.2015.16

mechanisms of automated porting; we had to elaborate a number of basic mechanisms manually, without using the built-in generator.

II. Memory Access Optimizations

Optimizations of memory access are among most effective. They are firstly aimed at increasing such characteristic of translated programs as memory access locality. These optimizations are, for example, various global data transformations:

- Structure field reordering,
- Structure splitting,
- Unused fields removing,
- Transforming array of structures into structure of arrays (AoS→SoA).

In some benchmarks, such as 429.mcf, 462.libquantum or 179.art, these changes can lead to performance gain of 50% or even 70%. For their proper application, you need to analyze all program cases of possible structures of a similar type because structures may sometimes form recursive dependencies, like, for example, in 429.mcf.

A more complicated transformation is linearization of arrays. This transformation is not very common: in both testsuites SPEC/CPU2000 and CPU2006 it occurs only in 183.equake; although the performance gain is 100%. However, the analysis, identifying the situation of multi-level memory allocation, allowed proving the independence of references to arrays for loop parallelization in 464.h264ref.

All of the above-mentioned optimizations require compiling in the “whole program” mode and conducting complex inter-procedural analysis of pointers [6].

One more technique, which belongs to the memory optimization class but does not involve global program data transformation, is data prefetching. Most modern microprocessor architectures have special instructions that can put data into cache before they are actually required during the execution of the program code, thereby reducing cache misses; the ARM and MIPS architectures are not exceptions. The compiler task here is to recognize a memory access, which can cause a cache miss, and, if the address for memory access is determined, to insert pre-fetch instructions at this address. The effectiveness of this optimization is quite dependent on architecture characteristics and memory sub-systems. In 470.lbm, the performance gain for Cortex-A15 Chromebook is 30%, for Aarch64 – 18%, and for MIPS – just a few per cent. Data prefetching, unlike global transformation of data structures, does not require inter-procedural context.

III. Auto-Parallelizing

For multi-core processors, auto-parallelization is, perhaps, the most effective high-level optimization. Its essence is in distribution of sequential program execution among multiple cores. There is sectional and cyclic parallelism. Sectional parallelism is the execution of consecutive pieces of code (or sections). It is justified only when these code sections are executed for a long time. On the one hand, it is a very rare situation, on the other hand, the analysis necessary for determining the correctness of this transformation can be very difficult.

Cyclic parallelism gives possibility to execute different logical loop iterations in separate execution threads. For a proper transformation, it is necessary to prove the absence of data dependencies between loop iterations called *cross-iterative dependencies*. The transformations aimed at eliminating these dependencies are the essence of preparations for the auto-parallelization of loops. One of cross-iteration dependencies types is recurrence, a dependency where the value of variable at each iteration depends on its value at the previous iteration. Certain types of recurrences do not interfere with parallelizing. The simplest example of recurrence is an inductive variable of loop. Another example of acceptable recurrence is so-called *reduction*:

`r = r` α `a`, where α is an associative operation

In this case, you can make a copy of this variable for each execution thread and then apply the operation α to the values obtained from all threads. It is important to remember that arithmetic operations with floating-point numbers have no associativity in the general case. Therefore, the parallelization of a loop that contains such dependences can be performed by a floating arithmetic operation only with the "fast-math" option which is usually disabled by default. Enabling this option allows achieving a significant speedup in 435.gromacs, 465.tonto and 482.sphinx3, for example.

Besides, there is so-called *swap-recurrence*. It means that the values of paired variables swap at each iteration, usually they are the addresses of arrays. Such dependency does not allow parallelizing a loop. However, recognition of this pattern can prove that the accesses to these two arrays are independent at each iteration which, in its turn, gives opportunity to parallelize internal loops, e.g. in the benchmarks like 183.equake or 470.lbm.

Cross-iterative dependencies for memory arrays (aggregate objects) are most difficult to analyze. In the considered loop, you have to check each pair of memory access operations. Any pair of operations can have three variants of relations:

1. They always access different objects (i.e. they are independent) and, therefore, do not disrupt parallelization.
2. They always access one and the same object.
3. Detecting the access is problematic.

In the variant three, you may insert the dynamic check of access independency to the translated code. It requires determining the change ranges of accessed addresses and is often non-trivial. For the variant two, there may be multiple patterns, similar to scalar. Those are reduction, private, first- and last- private arrays. In such cases, every execution thread allocates memory for a local array copy. Initialization occurs in the cases of first-private and reduction arrays; for the latter there is an additional collection of values from all iterations at the end. For last-private arrays, there is a restoring of values from the last iteration.

It is not difficult to recognize reduction arrays: you find writing and reading to the same address and data-flow between with operations. Then you have to prove that this data-flow comes to a reducible operation. For other cases, it is necessary to prove that reading from any memory cell occurs after writing to the same cell at the same iteration (private array case) or writing to this cell, during the loop, does not occur at all (first-private case). If an array can be accessed after a loop, it becomes last-private. The proof should be done by building a complete predicate for each access operation, as well as for access ranges. The obtained relations are complemented with the information coming from front-end about feasible (or unfeasible) values overflows during the execution of the operations, whereupon the task takes the form of Diophantine equations and inequalities system. To solve it, there is a tool of automated theorem proving z3 developed by Microsoft.

To allocate memory for the local copies of arrays we need to know their size. The ideal case is to find the range of address changes in the loop; and it is necessary to take into account the predicates followed by memory access. Another option is to use the information of the size at the point of array allocation; it is possible for either the arrays allocated in the same function as that of the considered loop, or for global arrays. The disadvantage of this method is that the array size may significantly exceed its part used in the loop. The overhead for initialization and recovery will be great then. Besides, reduction, first- and last-private arrays require precision in the size determination, because during array initialization and recovery you can go beyond the original array. For private arrays, upper estimate is enough.

It may be also that none of the described methods is suitable for the array size determination. Let us assume, an array comes as a function parameter and the indexing through it is indirect, like in 435.gromacs, 465.tonto and 416.gamess. For such cases, there are two strategies. The first one is allocating an array of a small size and, if necessary, its subsequent reallocation with data copy. Here the advantage is a low overhead, the disadvantage – the impossibility to evaluate the exact cost for array rebuilding; there are real examples where this strategy leads to a substantial performance regression. Moreover, an extra code inserted into the original loop can interfere with the subsequent vectorization of internal loops (e.g. in 435.gromacs). The second strategy is the preliminary loop run which counts only address ranges only in order to define array sizes precisely. The overhead can be substantial here, but the size of the needed array is well determined prior the parallelized loop.

Now, after all the analysis has been carried out and the possibility of parallelization has been proved, we create two loop versions, as well as the condition which contains dynamic checking of parallelization heuristic. It is the condition to determine which loop version is to be executed. The heuristic job is to try to evaluate the time of the original loop execution and its parallelized version considering the entire overhead.

According to the above, real-task loop parallelization requires implementing a variety of transformations supporting context preparation. During program execution, these transformations often entail such overhead that performance gets worse instead of improving. Therefore, fine-tuning of heuristics is obligatory for any transformation. The characteristics of specific target architecture should also be taken into account.

First of all it concerns inline substitution that greatly simplifies the context and reduces the analysis to intra-procedural. However, substitution to the corresponding loop is not always suitable for the heuristic. For example, 168.wupwise uses the library blas. The library functions are very large, but the substitution of specific parameters values can make them shrink to a few lines. Basic heuristics prevent substitution, so you need to carry out inter-procedural constant propagation with subsequent function cloning for every call context.

Another transformation to refine context is code replication, which allows determining important variables values (e.g. function pointers or predicates) due to the duplication of code. Thus, the benchmark 464.h264ref has three copies created for the basic loop; each of them can be finally parallelized.

Dataflow transformations also play an important role in optimization; those include moving invariants beyond the loop, scalarization, removal of redundant memory accesses and loop optimizations. The mere application of loop fusion, for instance, doubles the performance in 462.libquantum.

IV. VECTORIZATION

Vectorization is similar to parallelization, but it happens at a lower level – the level of individual operations – and uses special vector extensions of modern processor architectures, so-called SIMD-instructions. They enable performing same operations on multiple data simultaneously. Modern compilers usually support vectorization of loops and vectorization at the level of basic blocks. To ensure the correctness of vectorization, the compiler is to prove that the transformation keeps all the data dependencies of the original program. Beside loop dependencies, vectorization can fail because of inconsecutive memory access, complicated control flow, as well as the instructions which have no vector analogues.

Most of implementations of loop vectorization either can vectorize the loop completely or, upon coming across one or several obstacles, fail to apply vectorization at all. Some compilers can perform partial vectorization (i.e. vectorize only a part of code instructions) when it is not possible to vectorize the other part or inadvisable for performance reasons. Partial vectorization is usually achieved by splitting a loop into its multiple copies; that leads to growing overheads for transfer of intermediate calculation data between the copies.

The UTL utilizes a combined approach: it creates a loop where a part of instructions is vectorized, and duplicates scalar instructions in the remaining part like in loop unroll. First, we build a dependency graph. The dependencies that have failed to break at the compiling phase are eliminated via adding dynamic checks. After

that, there occurs the vectorization of instructions in the dependency graph nodes, and pack/unpack nodes are inserted at the interface between the scalar and vector instructions.

Based on heuristics, we carry out the optimization of the dependency graph; it determines which parts should be vectorized and which parts should be left in their scalar form. Using the strongly connected components of the dependency graph, we make the vectorized version of the loop in which the vectorized nodes are replaced by vector instructions and the scalar nodes are duplicated to match the amount of the data processed in the vector code. The scalar loop remains unchanged; its purpose is to finish the aliquant iterations of the transformed loop copy or the entire loop (in case the dynamic checks give evidence of a higher efficacy).

Combined approach helps to achieve significant results compared to standard approaches. In 435.gromacs, all memory accesses apply indirect indexing in the hot loop – that prevents their vectorization. However, vectorizing big amounts of arithmetic operations covers the overheads for packing the data read from the memory. It results 85.8% speedup in x86 and 26% in Aarch64.

Like auto-parallelization, vectorization requires simplifying the context in advance. In 456.hmmer, you have to apply the transformation of 'if-conversion' with subsequent peeling of the last iteration – to convert the loop to a single basic block. As a result, the vectorization of a part of the loop speeds up this benchmark by 87.3% in x86 and by 16% in Aarch64.

V. Performance Results

Testing of the developed SMCC compiler was conducted on two platforms – ARM and MIPS; their specifications, as well as the compared compilers, are presented in Tab. 1.

TABLE I. Hardware and software specification

	ARM	MIPS
Hardware	APM X-Gene XC-1 Mustang board Aarch64 Octa 1.6GHz, 8Gb RAM	Creator CI20 SoC: Ingenic JZ4780 CPU: Dual 1.2GHz XBurst MIPS32 little endian 32kI + 32kD per core 512K shared L2, 1Gbyte DDR3
Cores	8	2
Compilers	native gcc-linaro-4.9	gcc cross 4.9.2
	llvm 3.7	
Compiler options	-flto -O3 -ffast-math -mtune=cortex-a57 (only for gcc)	-flto -O3 -ffast-math

The comparison processes were carried out on the test suite SPEC/CPU2006. For the run on ARM, we used reference data. On the MIPS platform, the benchmarks used train data because reference data are too large for this platform. Besides, 434.zeusmp was excluded from testing on MIPS, as it requires over 1GB of memory for its own code.

As the LLVM compiler does not support front-end with FORTRAN, for the tasks written in this language, we utilized the DragonEgg plugin [9]. Its support was discontinued in LLVM starting the version 3.4; we have to support it on our own account.

Fig. 1 gives the comparisons of SMCC with Linaro GCC and LLVM correspondingly. The compiler options of all three compilers are similar for each of the platforms, see Tab. 1. On 8-core ARM platform, the performance gain of SMCC is 47.5% vs. Linaro and 55.7% vs. LLVM. On 2-core MIPS platform, SMCC wins them by 17.6% and 22.8%, see Fig. 2.

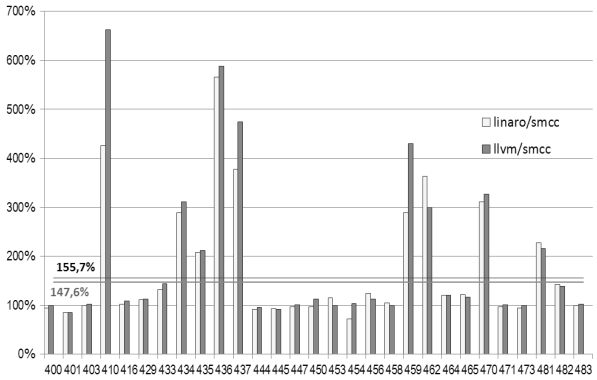

Fig. 1. smcc vs. gcc-linaro and llvm on 8-core ARMv8

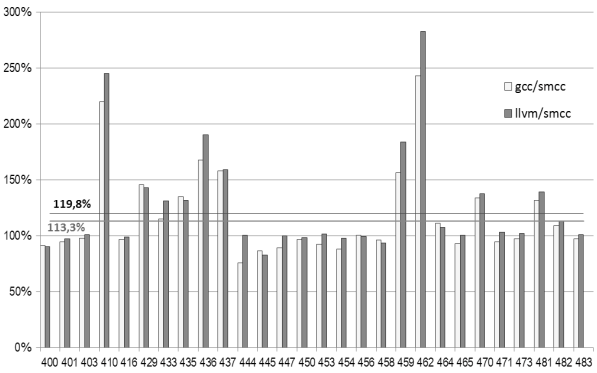

Fig. 2. smcc vs. gcc and llvm on 2-core MIPS

It is evident that the main driver for the gain is auto-parallelizing: the transformation is applied to 12 out of 29 tasks at the ARM platform, which gives from 50% to 500% of gain on 8 cores.

SMCC still shows growth in performance on one core: 3% vs. GCC and 5% vs. LLVM in ARM, 2,5% and 6% accordingly in MIPS.

In the benchmark 410.bwaves, one core – MIPS, SMCC outperforms both compilers on 30 and 44% correspondingly. The analysis of this unexpected fact showed that both compilers could not apply their standard transformation in this architecture:

```
pow(x,1.75) → x*sqrt(x)*sqrt(sqrt(x))
```

VI. LLVM Port to Lynx Processor

The company Milandr has developed a DSP processor called Lynx. This processor architecture is an enhanced architecture of TigerSHARC TS201 processor made by Analog Devices. It is notable for double precision floating-point calculations support [11]. There is only a proprietary compiler for TS201 and it is not available for modification. It was decided to create compiler for the Lenx architecture on the base of LLVM. To add a new back-end, the LLVM infrastructure uses a special language tablegen which allows describing

- Register file – the number of registers, types of stored data, encoding the register number in the operation code;
- Instruction set – coding instructions, their representation in the assembler, input and output operands, instruction semantics;
- Calling conventions – rules for the transfer of parameters to functions and return of the result.

By these means, the largest amount of code necessary for compiling the program for the target platform is generated by tablegen interpretive routine. The high degree of automation allows fast realization of minimal support for the new architecture and focusing on the implementation of specific optimizations.

However, it has appeared that Lynx does not give the opportunity to get all the code described for other architectures by tablegen. The assembly language syntax for Lynx has a peculiar property that precludes using standard LLVM syntax parser. The majority of Lynx instructions are written in the format that is close to those used in high-level languages – where they indicate an output operand first, then there is an assignment sign, input operands and an operation sign. For example, addition looks like follows:

```
J0 = J1 + J2;;
```

Such syntax does not fit into the conventional scheme, in which instructions are sequences of mnemonics and operand list. Unfortunately, tablegen turned out to be useless for implementing an 'assembly file to object code' converter.

Another serious issue was memory addressing scheme used in Lynx architecture. Unlike most platforms where addressing is carried out in bytes consisting of 8 bits, Lynx addressing unit is a 32-bit cell. In other words, a byte of this architecture consists of 32 bits, that does not contradict to the language standards of C. However, LLVM infrastructure is not intended for such scheme. Solving this problem requires making numerous amendments to the core-part of the compiler.

Conclusion

Creating a microprocessor which has no effective development tool risks bringing to naught all of the efforts undertaken by the hardware developers. You can invent your own compiler (like Intel) or refine an open solution in collaboration like community of ARM manufacturers. Anyway we must not ignore this aspect of work at all.

Currently all available multi-core ARM-based processors have symmetric architecture in terms of memory access (SMP). ARM manufacturers, targeted to the server market, have already started works on solutions like Non-Unified Memory Access (NUMA). To disregard this architectural feature means to lose in performance and, as a result, competitive advantages. For instance, Intel has a rich experience in developing NUMA-solutions and certainly considers these elaborations in their products for software products. Therefore new algorithms and heuristics that take into account architectural asymmetry should strengthen both memory access optimizing transformations and auto-parallelization.

The Milandr RC has also developed an integrated module based on its Lynx processor. It consists of five four-processor clusters. The speed of data exchange between processors greatly depends on whether they are in the same cluster or different.

Creating a common parallelizing solution for compiler which could be configured for various asymmetric computing systems is an interesting and important challenge. The introduction of such solution in the UTL is one of possible development lines for this library.

References

[1] GCC, the GNU Compiler Collection, http://gcc.gnu.org/

[2] The LLVM Compiler Infrastructure, http://www.llvm.org/

[3] Linaro GCC, https://launchpad.net/gcc-linaro/

[4] A. Yu. Drozdov, "Component Approach for Construction of Optimizing Compilers", Programming and Computer Software, 2009, Vol. 35, No. 5, pp. 291-300, 2009.

[5] A. Yu. Drozdov, S.V.Novikov, "A program auto-parallelizer based on the component technology of optimizing compiler construction", Programming and Computer Software, 2009, Vol. 35, No. 6, pp. 321-339, 2009.

[6] Drozdov A.Yu., Vladislavlev V.E., Inter-procedural points-to analysis // Informatsionnye tekhnologii, Application 2, Feb 2005.

[7] Drozdov A.Yu., Novikov S.V., Vladislavlev V.E., Kochetkov E.L., Il'in P.V., Program auto parallelizer and vectorizer implemented on the basis of the universal translation library and LLVM technology, Programmirovanie, 2014, Vol. 40, No. 3.

[8] SPEC, SPEC/CPU2006 testsuite, http://www.spec.org/cpu2006/

[9] DragonEgg plugin, http://dragonegg.llvm.org/

[10] Lubitzin V., High-speed multiclaster integrated data processing module, journal "Electronika: Science, Technology, Business", №6, 2015.

[11] Myakochin Yu., High-performance DSP processor for communication systems, «Components & Technologies», № 159, pp. 82-84, 2014.