SERBIAN JOURNAL OF ELECTRICAL ENGINEERING
Vol. 10, No. 1, February 2013, 91-101

UDK: 004.383/.384-181.4 ; 004.431.4 **DOI: 10.2298/SJEE1301091K**

A Solution for Automatic Parallelization of Sequential Assembly Code

Djordje Kovačević[1], Mladen Stanojević[1], Vladimir Marinković[1], Miroslav Popović[1]

Abstract: Since modern multicore processors can execute existing sequential programs only on a single core, there is a strong need for automatic parallelization of program code. Relying on existing algorithms, this paper describes one new software solution tool for parallelization of sequential assembly code. The main goal of this paper is to develop the parallelizator which reads sequential assembler code and at the output provides parallelized code for MIPS processor with multiple cores. The idea is the following: the parser translates assembler input file to program objects suitable for further processing. After that the static single assignment is done. Based on the data flow graph, the parallelization algorithm separates instructions on different cores. Once sequential code is parallelized by the parallelization algorithm, registers are allocated with the algorithm for linear allocation, and the result at the end of the program is distributed assembler code on each of the cores. In the paper we evaluate the speedup of the matrix multiplication example, which was processed by the parallelizator of assembly code. The result is almost linear speedup of code execution, which increases with the number of cores. The speed up on the two cores is 1.99, while on 16 cores the speed up is 13.88.

Keywords: MIPS, Compiler, Assembler, Parser, SSA form, Data flow graph, METIS, Linear scan register allocation.

1 Introduction

The appearance of multiprocessor systems has provided new opportunities to increase the speed of program execution. Parallelization of existing sequential programs and the generation of new parallel programs have become important areas of research. The aim of this work is to develop parallelizator of assembly code and evaluate how parallelization affects the speed of program execution. Besides the large set of programming languages, there is now a considerable choice in their upgrades in the form of libraries (MPI, OpenMP, OpenCL, etc.) that allow programmers to develop applications that run

[1]Faculty of Technical Sciences, University of Novi Sad, Trg Dositeja Obradovića 6, 21000 Novi Sad; E-mails: Djordje.Kovacevic@rt-rk.com, Mladen.Stanojevic@rt-rk.com, Vladimir.Marinkovic@rt-rk.com, Miroslav.Popovic @rt-rk.com

concurrently on multiple processors or multicore processor of a multiprocessor computer. Information technologies have gone a step further, separating themselves from the physical support to the higher layers of abstraction of computer systems. This primarily applies to the languages and technologies with an automatically controlled allocation of memory (Microsoft C# with .Net, Java, etc.). The development of smart phones, tablets, smart TVs, and their growing popularity led to the return to massive use of low level programming languages (Assembler and C). Certain efforts were made for automatic parallelization of sequential C code [1]. Still, increasing the use of C/C++ programming languages, what is also the contribution of the new C++11 standard, there is a need for assembler programming for real time processing. RISC (Reduced Instruction Set Computing) processors, which are mostly used in phones and similar embedded devices, are starting to follow the CISC (Complex Instruction Set Computing) processors. Motivated by the increasing popularity of RISC processors, this paper, relying on the existing algorithms, implements the translator of source code that automatically translates a given sequential code for execution on a single core to parallel code intended for a fixed number of cores. A number of papers have already made different researches on techniques of assembly code optimization, but not parallelization. Amme *et al.* [2] described a new approach for the determination of data dependences in assembly code. It is based on a sophisticated algorithm for symbolic value propagation, and it can derive value-based dependences between memory operations instead of just address-based dependences. It was integrated into the system for assembly language optimization and proved to be a good improvement in terms of the precision of the dependence analysis in many cases.

The techniques used in this paper are taken from the design of program compilers. The techniques are adapted to a specific problem, namely: static single assignment form [3] (hereinafter SSA form) that is intermediate representation of data during the program translation, and the linear scan algorithm for register allocation as the phase that follows parallelization algorithm [4].

Section 2 describes the architecture of parallelizator and presents activity diagram and class diagram. Parallelizator of assembly code is divided into three parts: the frontend, middle, and backend. The frontend of parallelizator, described in Section 3, represents parser of assembler code with the transformation to SSA form. Section 4 describes the middle part of parallelizator that contains the data flow graph and the parallelization algorithm. The backend of parallelizator, which is used for both register allocation of processed assembler code and code emitter, is given in Section 5. Section 6 refers to the parallelizator evaluation, while the last section presents the conclusion.

2 The Architecture of Parallelizator

The main assignment of parallelizator is to read sequential assembler code for execution on one core at the input stage, and to deliver parallelized code for MIPS processor with multiple cores at the output.

Fig. 1 shows the activity diagram of parallelizator, which illustrates the following sequence of events:

Assembler code was obtained by compiling C code with GCC compiler. By including option –S in the GCC compiler, instead of machine code, the assembler code is produced at the output.

The parser translates input assembler file into programming objects suitable for further processing. These objects imitate normal intermediate code of programming interpreter. The object contains pointers to the registers which it is related to. Registers are represented as variables for transition to SSA form.

Unlike the original source code that usually redefines each variable more than once, in the SSA form each variable is defined exactly once, and can be used multiple times [3]. The realization of SSA form produces lists of used and defined variables, which are used to obtain intermediate code that is easily to handle later on.

1. Data flow graph is determined over all instructions of assembly code, where each instruction has a list of instructions that it depends on.
2. The main idea for parallelization was the partitioning of data flow graph as studied in [5, 6]. Capko *et al.* [5] stated that the necessary preconditions for the efficient calculation are optimal load balancing of processors and data model partitioning among processors. They proposed the novel multilevel Super-Roots algorithm that is actually an improved existing algorithm. Existing algorithm, METIS [7] from METIS from METIS tool was used for initial partitioning of data model. The proposed algorithms are applied on data model described in [5]. They stated that experiments have shown that Super-Roots algorithm achieves better results than METIS multilevel algorithm in many cases, but this is true only in the case when calculation regions are weakly connected. In their next paper [6], Capko *et al.* included the solution for partitioning dynamically changed graphs. Based on the data flow graph, the parallelization algorithm divides instructions to different cores. After the parallelization algorithm is performed, it is possible that the liveness of a variable is discontinued, i.e. the variable continues to live on another core. In that case it is necessary to add synchronization instructions (load/store instructions) to the cores.
3. The classic compilers use algorithm for register allocation using mathematical theory of graph coloring. The biggest drawback of this method is that it takes a lot of processing power to perform. Because of

its speed, which is up to 70 times higher than the graph coloring [4], the linear register allocation is used. When performing a linear registers allocation after SSA analysis, it is necessary to make the liveness analysis as an intermediate step. This analysis is achieved by only a single pass through the list of instructions, where the positions of variables in the SSA form are sufficient information.

Registers allocation is shown in **Table 1**.

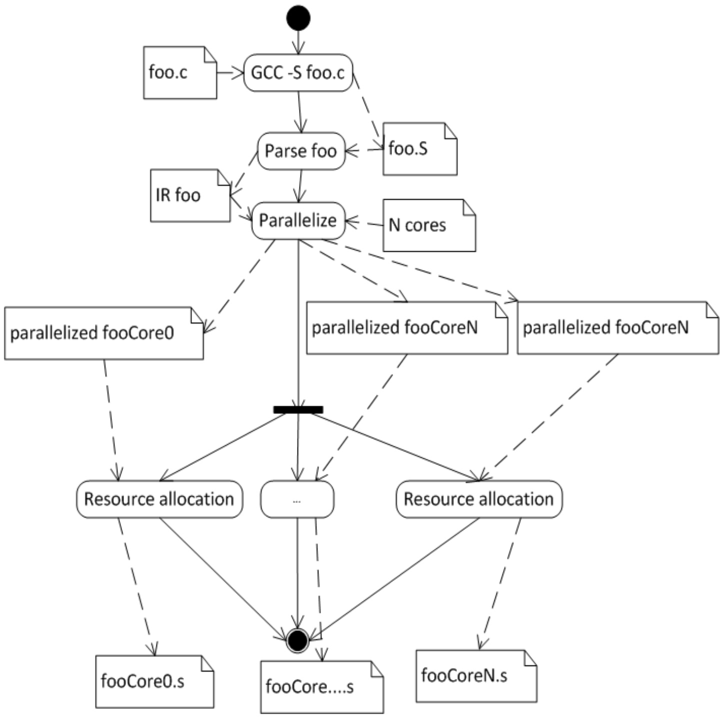

Fig. 1 – *UML activity diagram* of *parallelizator*.

Table 1
The registers allocation flow, from the original program to the registers of hypothetical processor.

	1	2	3	4
Source code	X = 2	Y = 3	X = X +Y	X = X + 5
SSA	X.0 = 2	Y.0 = 3	X.1 = X.0 + Y.0	X.2 = X.1 + 5
Live variables	Rs1	Rs1, Rs2	Rs1, Rs2, Rs3	Rs3, Rs4
Register allocation	R1	R1, R2	R1,R2,R3	R1, R3

Fig. 2 shows modules (with appropriate classes) of parallelizator: parser (CParser), module for transitioning code to SSA form (CRegisterRename), the data flow graph builder (CDdg), METIS library for separating data flow graph to the desired number of partitions (CDDGPartitioner), module for data synchronization based on shared memory (CParallelizer), registers allocation (CResourceAllocator) and code emitter module (CDumper).

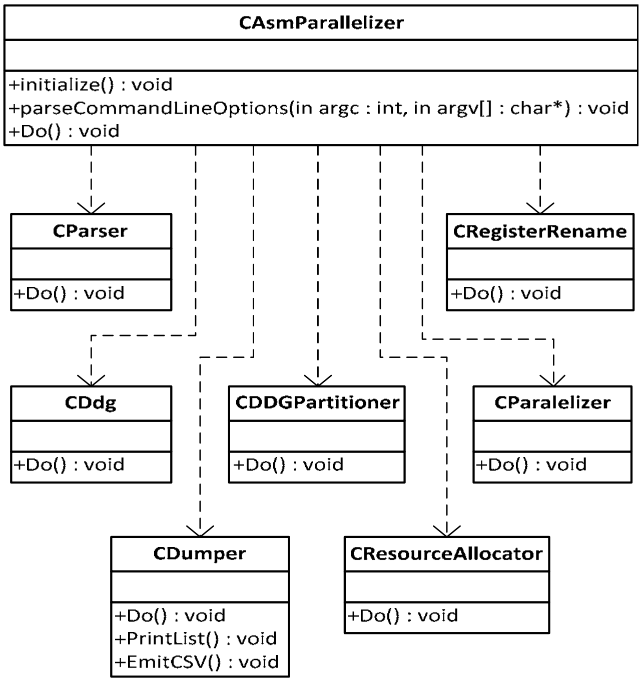

Fig. 2 – UML *class diagram* of *parallelizator.*

The program was implemented so that the main function calls Do() method of class CAsmParallelizer. The Do() method calls Do() methods of remaining classes that implement already mentioned modules of parallelizator.

3 Parallelizator Frontend

Parallelizator frontend consists of assembly code parser that performs the transformation to SSA form. The parser is implemented by string tokenization and switch-case decision structure. It has two levels:

1. The first level extracts line by line from the assembly code, ignoring lines that are not of interest for further processing.
2. The second level divides the text line into tokens of interest for further processing.

For example, the line "ADD $t1, $t2, $t3" is divided by the parser to 4 symbols: "ADD" – type of instructions, "$t1" – the destination register, "$t2" – source register and "$t3" – source register.

The condition for switch-case structure is a type of instruction. Switch-case structure recognizes the type of instruction and sets lists of uses and definitions of variables.

While extracting lines from an assembly code line-by-line, the parser classifies the whole assembler code to: program, functions and basic blocks. All the data elements are connected by the pointers. The initial assembler program is presented as a list of objects CFunctionBlock, where each single element of the list contains information about one function of the initial program. The function is considered to begin with the directive .align and to end with the directive .size. CFunctionBlock contains information about basic blocks, specifically a list of objects of type CBasicBlock, while the basic block contains information about the instructions, namely a list of objects of type CInstruction.

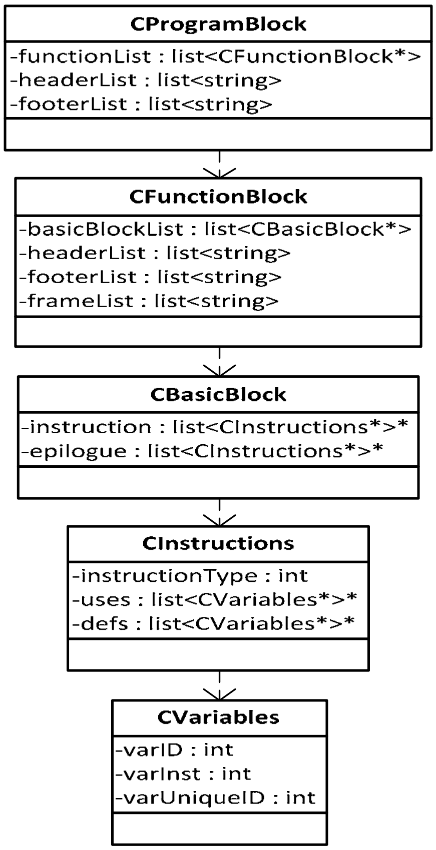
CProgramBlock
-functionList : list<CFunctionBlock*>
-headerList : list<string>
-footerList : list<string>
CFunctionBlock
-basicBlockList : list<CBasicBlock*>
-headerList : list<string>
-footerList : list<string>
-frameList : list<string>
CBasicBlock
-instruction : list<CInstructions*>*
-epilogue : list<CInstructions*>*
CInstructions
-instructionType : int
-uses : list<CVariables*>*
-defs : list<CVariables*>*
CVariables
-varID : int
-varInst : int
-varUniqueID : int

Fig. 3 – *The data structure of* the *assembly code parser.*

Each instruction contains the type of instruction, and two lists of variables: (1) list of variables used by the instruction, and (2) list of variables defined by the instruction.

Data structure of the assembly code parser is illustrated in Fig. 3.

The module for transitioning code to SSA form is part of the parser, but as an important technique, it had to be presented separately. SSA formed code implies that each variable is defined exactly once, and can be used multiple times. These variables are enabling easier implementation of optimization techniques in the program compilation process.

There are papers that show that the linear scan algorithm for register allocation on SSA form provides the same or even better results than the classical register allocation via mathematical methods of graph coloring [4]. In the example given in **Table 2**, the variable X based on SSA analysis is renamed in X.1, as the result of variable redefinition. The example shows that X.2 does not take the value X.0, but that it takes the value X.1 since X is redefined.

4 Parallelizator Middle Part

The middle part of parallelizator consists of the following modules: the data flow graph, METIS library for dividing data flow graph to the desired number of partitions, and modules used for data synchronization via shared memory.

The structure dependencies that exist in the program are well formed using the graph as the program abstraction. Data flow graph is performed over all instructions of assembly code. The nodes of the graph represent instructions, while branches of the graph connect dependent instructions. Branches of the graph represent variables, thus instruction dependencies are defined according to use of same variables. Each instruction has a list of instructions from which it depends on. To determine the relationship between instructions, variables of each instruction are observed. Two instructions are dependent if same variable appears in both instructions.

Based on the data flow graph, the parallelization algorithm divides instructions into different partitions, intended to fit to different cores. In the case of data flow graph described in this paper, calculation regions are strongly connected and the graph is not changed during partitioning, i.e., after compilation. Thus, the reduced algorithm from those from [5] and [6] is used. Specifically, multilevel k-way algorithm for graph partitioning from METIS tool is selected [8]. This algorithm divides the data flow graph into a desired number of partitions, providing approximately the same load per each core of processor, but also trying to find the least possible number of links between partitions on which it divides the graph.

After the instructions are distributed by the cores, the synchronization instructions are added where necessary (load/store instructions). If there is a variable in the instruction which is used in one core, and it is not defined in that core, it is necessary to add synchronization instructions to the cores. The store instruction should be added to the core where the variable is defined and the load instruction should be added to the core where the variable is used. Specifically, the store instruction is added after the variable is defined in some core, while load instruction is added before using the variable in the other core.

5 Parallelizator Backend

The backend of parallelizator performs resource allocation. Resource allocation includes the allocation of physical locations, registers and memory to logical data, in this case, the data presented in the SSA form.

In case of the parser model, SSA form considers one exception. The registers that should not be changed present that exception. Changing any of these registers would introduce undefined program execution as a result. Registers which must not be changed are: zero register ($0), register that assembler uses to switch from pseudo-instruction to physical instruction ($at), registers reserved for the OS kernel ($k0, and $k1), a global pointer ($gp) the stack pointer ($sp), the frame pointer ($fp), return address register ($ra), registers that hold return value from a function ($v0, and $v1), and registers for passing parameters to function ($a0, $a1 , $a2, $a3) [9].

All other registers are included in the set of registers available for allocation. Type of this set is the STL set. The uniqueness of the registers was accomplished using the set. As such, they are free for use during allocation. When the register is allocated, it is transferred from a set of available registers to the set of occupied registers. When the register lifetime comes to an end, according to liveness analysis data, the register is returned to the set of available registers.

6 Evaluation

Evaluation was done on the example of matrix multiplication.

The dependency between the source code speedup and the number of cores was observed in this example. The speedup is defined as the ratio of Ts/Tp, where Ts is execution time of sequential program, while Tp is execution time of parallel program. It is assumed that all instructions have the same execution time. Ts presents the total number of sequential assembly code instructions, while Tp is the largest number of distributed instructions per core.

Table 2 shows how number of cores influences the assembly code execution speedup. In the example of matrix multiplication, the code execution speed is growing, as the number of cores increases. Using only two cores the speedup is doubled compared to the sequential code execution. Using 16 cores, the speedup is almost 14 times larger.

Table 2
Results of experimental measurements of code execution speedup.

Number of cores	Speedup code on the example of matrix multiplication
2	1.99448
4	3.94536
8	7.76344
16	13.8846

Fig. 4 presents graphical results of the code execution speedup depending on the number of cores.

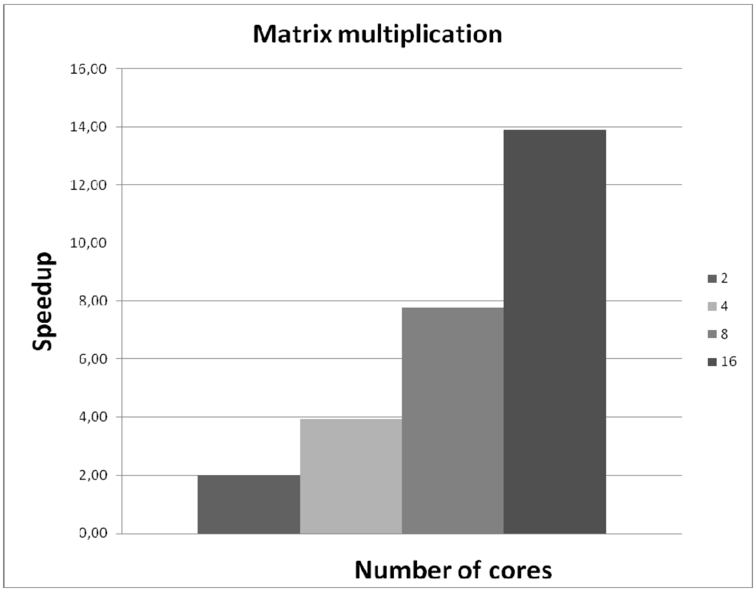

Fig. 4 – *Code execution speedup results depending on the number of cores on the example of matrix multiplication.*

7 Conclusion

This paper presents a new solution for automatic parallelization of assembly code. It is separated in three parts: frontend, middle part, and backend.

Most important methods included in proposed assembly code parallelization are transition for single static assignment from frontend, building data dependency graph and graph partitioning from middle part, and finally resource allocation and instructions scheduler from backend.

By analyzing the testing results it can be concluded that the results are satisfactory. High and almost linear code execution speedup was achieved when increasing number of cores. The speedup for 2 cores is 1.99, while for 16 cores the speedup is 13.88.

Current solution builds data dependency graph according to register dependency only, i.e. the memory locations dependencies are not taken into account. In order to avoid the risk of data races, alias analysis in terms of memory locations should be added to this solution. Alias analysis of executable code is well studied topic [9], but these new dependencies will definitely make graph partitioning more complex and implicitly reduce partitioning effectiveness and code execution speedup accordingly.

The second limitation of this solution is partitioning of graphs generated from sequential code only. In order to make it applicable for code structured in any way, analysis based on control flow graph and the determination of dependencies among basic blocks should be added.

The choice of the linear registers allocation in this paper has proved as a relatively simple algorithm that does not consume a lot of resources for allocation. In terms of allocation efficiency it is not behind the conventional algorithm for registers allocation, as stated in [10].

Having in mind that other solutions are only offering different techniques for optimization of assembly code, the method proposed in this paper is a first step towards an ambitious goal of making a complete translator of sequential assembly code to the parallel one.

8 References

[1] N. Vranic, V. Marinkovic, M. Djukic, M. Popovic: An Approach to Parallelization of Sequential C Code, 2nd Eastern European Regional Conference on the Engineering of Computer Based Systems, Bratislava, Slovakia, 05 – 06 Sept. 2011, pp. 143 – 146.

[2] W. Amme, P. Braun, F. Thomasset, E. Zehendner: Data Dependence Analysis of Assembly Code, International Journal of Parallel Programming, Vol. 28, No. 5, Oct. 2000, pp. 431 – 467.

[3] A.V. Aho, M.S. Lam, R. Sethi, J.D. Ullman: Compilers, Principles, Techniques and Tools, 2nd Edition, Pearson Education, Boston, MA, USA, 2007.

[4] C. Wimmer, M. Franz: Linear Scan Register Allocation on SSA Form, Department of Computer Science University of California, Irvine.

[5] D. Capko, A. Erdeljan, M. Popovic, G. Svenda: An Optimal Initial Partitioning of Large Data Model in Utility Management Systems, Advances in Electrical and Computer Engineering, Vol. 11, No. 4, Nov. 2011, pp. 41 – 46.

[6] D. Capko, A. Erdeljan, G. Svenda, M. Popovic: Dynamic Repartitioning of Large Data Model in Distribution Management Systems, Electronics and Electrical Engineering: System Engineering, Computer Technology, No. 4 (120), 2012, pp. 83 – 88.

[7] G. Karypis, V. Kumar: A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs, SIAM Journal of Scientific Computing, Vol. 20, No. 1, Aug. 1998, pp. 359 – 392.

[8] G. Karypis, V. Kumar: Multilevel k-way Hypergraph Partitioning, 36th Design Automation Conference, New Orleans, LA, USA, 21 – 25 June 1999, pp. 343 – 348.

[9] D. Sweetman: See MIPS Run, 2nd Edition, Morgan Kaufman, San Francisco, CA, USA, 2007.

[10] S. Debray, R. Muth, M. Weippert: Alias Analysis of Executable Code, 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Diego, CA, USA, 19 – 21 Jan. 1998, pp. 12 – 24.

[11] M. Puletto, V. Sarkar: Linear Scan Register Allocation, ACM Transactions on Programming Languages and Systems, Vol. 21, No. 5, Sept. 1999, pp. 895 – 913.