# An Automatic Instruction-Level Parallelization of Machine Code

Vladimir MARINKOVIC[1,2], Miroslav POPOVIC[1], Miodrag DJUKIC[1]
[1]*University of Novi Sad, Faculty of Technical Sciences, Novi Sad, Serbia*
[2]*RT-RK Institute for Computer Based Systems LLC, Novi Sad, Serbia*
*vladimir.marinkovic@rt-rk.uns.ac.rs*

***Abstract*—Prevailing multicores and novel manycores have made a great challenge of modern day - parallelization of embedded software that is still written as sequential. In this paper, automatic code parallelization is considered, focusing on developing a parallelization tool at the binary level as well as on the validation of this approach. The novel instruction-level parallelization algorithm for assembly code which uses the register names after SSA to find independent blocks of code and then to schedule independent blocks using METIS to achieve good load balance is developed. The sequential consistency is verified and the validation is done by measuring the program execution time on the target architecture. Great speedup, taken as the performance measure in the validation process, and optimal load balancing are achieved for multicore RISC processors with 2 to 16 cores (e.g. MIPS, MicroBlaze, etc.). In particular, for 16 cores, the average speedup is 7.92x, while in some cases it reaches 14x. An approach to automatic parallelization provided by this paper is useful to researchers and developers in the area of parallelization as the basis for further optimizations, as the back-end of a compiler, or as the code parallelization tool for an embedded system.**

***Index Terms*—parallel architectures, parallel programming, multicore processing, assembly, processor scheduling.**

## I. INTRODUCTION

In modern days, all segments of the computer market, from powerful compute-servers to handheld devices, involve multiprocessor systems. The introduction of these systems has provided opportunities to increase the speed of program execution or to save energy keeping the old program execution time. Hence, emerging multicores and even manycores make parallel program execution essential. Nevertheless, most of embedded software is still written as sequential and transparent support for this practice demands the parallelization of these sequential programs. Parallelization could be done (a) manually, (b) automatically programmer directed, and (c) fully automatically.

Forcing the program to be executed on several cores simultaneously by hand is a complicated, error-prone, exhausting, time consuming, and iterative process. It assumes switching to threads model (e.g. POSIX Threads), which requires a lot of changes in the source code, especially in the case when we already have a sequential program. It is more appropriate when writing a parallel program from the scratch. Yet, a programmer has to think parallel and write the program in that way, although it results in significantly lower productivity per line of code compared to serial programming [1]. On the other hand, it can give the best results in terms of speedup.

Automatic but programmer directed method usually implies only minor changes in the code (e.g. adding compiler directives or compiler flags, using existing templates, etc.) although, unfortunately, it is not always that simple. Generally, it is better for parallelization of existing sequential code than doing it by hand. However, it gives worse results regarding speedup. The most prominent approaches for automatic but programmer directed parallelization of sequential codes are: Intel Threading Building Blocks (TBB) [1-2], Intel Cilk Plus [2], Open Multi-Processing (OpenMP) [3], Open Computing Language (OpenCL) [4], etc. All these approaches contribute to satisfactory speedup. Some of these runtime libraries were analyzed by [5]. The authors showed that TBB runtime library can contribute up to 47% of the total per-core execution time on a 32-core system, whereas OpenMP gives even higher overheads, due to the lock contention. Simple parallelization is possible using existing templates, but only in specific applications. In a general case, e.g. irregular applications, parallelization is much more complicated. The programmer needs to notice places suitable for parallelization and to introduce parallelization by making non-trivial changes in the code.

The fully automatic approach is still an open issue. It should enable the parallelization of sequential code, with no need for changing the source code. This may result in slightly lower speedup but this is acceptable considering the significant benefit of the process automation.

This paper considers automatic code parallelization, specifically, it deals with developing a sequential code parallelization tool at the binary level that produces parallel program, and the validation of this approach. Beside automatic parallelization, the essential ability of this kind of tool is balancing the load of all cores. To the best of our knowledge, there are only a few methods regarding automatic parallelization at the binary level. Moreover, there is no tool for automatic parallelization of irregular programs at binary level. Mostly, these tools cover static parallelization of loops with or without loop-carried dependencies, and dynamic parallelization of loops and recursive techniques. Kotha *et al.* [6] presented a method for automatic parallelization inside a binary rewriter. It is a rare method that focuses on static parallelization of sequential code at the binary level, an approach that resembles the method we propose here. Unlike our method, the authors look at loop-level parallelism, while our goal is the Instruction-Level Parallelism (ILP) of all (affine as well as non-affine) programs. Our work can be considered to be the

This work was partially supported by the Ministry of Education and Science of the Republic of Serbia, Grant TR 32031.

Digital Object Identifier 10.4316/AECE.2018.01004

follow-up or generalization of the work of Kotha *et al*. as they have made the first step in the research field of parallelization of binaries, but for very specific problems.

The machine code at the binary level is taken as the input of the parallelizator since it contains all the details regarding target architecture. Even though the room for maneuver regarding machine independent optimizations and macro optimizations is reduced, since the binaries lack the high-level information, all the information for ILP and machine dependent optimizations is gained at this level. On the other hand, machine independent optimizations are already considered by the modern compiler (e.g. gcc), hence the presented parallelizator could be successfully used as the separate back-end stage of the same compiler. In this method, it is proved that the parallelization can be done on binary level without array index or symbolic information. Good load balance is achieved by a novel ILP algorithm which is developed for assembly code. It uses the register names after Single Static Assignment (SSA) to find independent blocks of code and then schedules independent blocks using a set of tools for partitioning graphs called METIS [7] algorithm as the start point improved by several strategies afterwards. Furthermore, according to Kotha *et al.* [6], beside the already mentioned ones, the parallelization at the binary level has several advantages over parallelization at the intermediate level (inside a compiler): (1) it works for all programming languages and compilers; (2) there is no need for software toolchain replacement; (3) it has high portability since the tool made for one architecture could be used for any compiler; (4) there is no need for source code in order to parallelize a binary; (5) it can be used for assembly code directly; and (6) beside a software developer, it can be used by the end user, as well.

Another problem in parallelization is the validation of it. Program execution time measurements on both platform simulation model and target architecture are considered. A similar approach is a multiprocessor hardware simulation model used by Wang *et al.* [8]. The speedup, which is defined as the ratio of execution times of a sequential program and a parallelized program, is taken as the performance measure in the validation process. Also, a performance model is defined as estimating the program execution time, and is verified by comparing the estimated time to the measured values afterwards. Optimal load of all processor cores enables significant speedup for regular and irregular applications, and, most importantly, without changing the source code of sequential program.

The scope of application of the proposed parallelizator is a range of multicore embedded platforms, primarily used for hard real-time applications. For selecting the specific target architecture, additional elements have to be considered. According to Vishkin [9], in contrast to serial computing, parallel computing has never successfully used a general-purpose programming model. The main reason for that is the cache coherence. The presence of cache or virtual memory in a deterministic system makes it nondeterministic and significantly reduces worst-case execution time. Consequently it is not acceptable for hard real-time systems. Furthermore, Vishkin [9] states that parallel computing has to use domain-specific models and architectures with local parallel memories and without cache, not only for real-time systems, but for any specific domain. Taking the same attitude, the parallelizator targets symmetric multicore embedded platforms without cache memories. It is currently developed in two branches: (a) the extension of previously developed optimized C-compiler for multicore DSP [10], and (b) the post-build tool for multicore RISC and domain-specific architectures (e.g. MIPS, MicroBlaze, etc.), used after existing C-compiler, i.e. gcc.

The rest of the paper is organized as follows: next section covers related work. Section 3 describes the proposed method for parallelization of assembly code, whereas specific case study with target architecture is given in Section 4. Section 5 presents evaluation with results. Section 6 outlines concluding remarks with future challenges.

## II. Related Work

This section lists and discusses the work related to (1) Static Automatic Parallelization Methods, (2) Dynamic Automatic Optimization Methods, (3) Parallelization Methods for FPGA Processor Prototypes, and (4) Graph-Partitioning Based Optimizations. The supportive related work is also given throughout the paper.

### *1) Static Automatic Parallelization Methods*

In terms of automatic parallelization of sequential C code, significant effort was made by Vranic *et al.* [11], who propose two algorithms as the main part of parallelizator added to C compiler that targets modern DSP. They use front-end of existing compiler to produce the intermediate representation (IR) of the code, transformed to *N* partitions, and returned to compiler's back-end which produces *N* executables later on. That approach shows speedup of even 7.96x for 8-core processor in applications with low data dependencies (only shown on the simulation model of platform which seems to be unrealistic). The approach considers only the code without branches and loops, does not include alias analysis, and it is inapplicable for most C problems. Consequently, the approach is not very sound, even though it provides a good basis for a complete solution of automatic parallelizator that produces high speedups.

Assembly code proves to be suitable for parallelization producing almost linear speedup with some benchmarks, according to Kovacevic *et al.* [12]. The authors use gcc to produce assembly code, and then load the assembly code to parallelizator based on Data Dependency Graph (DDG) partitioning using METIS algorithm. Still, the parallelizator can only deal with code without branches and loops. Also, memory is not taken into account when the DDG is made.

There are several efforts regarding automatic compiler optimizations and parallelization by improving data analysis with special emphasis on loop optimization. Kyriakopoulos *et al.* [13] try to find new data dependence analysis techniques taking nonlinear expressions into account. In that way they manage to handle complex instances of the dependence problems and to increase program parallelism. Consequently, parallelization results are improved giving higher speedups with some benchmarks. We support the parallelization of any type of loop in the way of parallelizing loop body (also called DOACROSS parallelism) which is better than parallelizing independent loop iterations only (also called DOALL parallelism) [14].

Ottoni *et al.* [14] provide solution called Decoupled

Software Pipelining (DSWP) that exploits fine-grained pipeline parallelism that successfully extracts long-running, concurrently executing threads. Author's fully automatic compiler implementation demonstrates significant gains. Still, it is validated on dual-core processor simulation model only. DSWP uses Thread-Level Parallelism (TLP), specifically DOACROSS loop parallelism. In contrast, our approach uses ILP, including DOACROSS loop parallelism. Furthermore, it is scalable and verified on target platform.

Campanoni *et al.* [15] describe HELIX, the technique for fully automatic loop parallelization that assigns successive loop iterations to separate threads. Interestingly, the authors mitigate loop-carried data dependencies by code optimization introducing helper threads for prefetching synchronization signals. HELIX achieves significant speedups using TLP and DOALL loop parallelism for optimized loops on source level, with slight inefficiency caused by inserted inter-thread communication. Our approach uses ILP of machine code and DOACROSS loop parallelism without aforementioned overhead.

Dave *et al.* [16] make an apparent advance to automatic parallelization of sequential C code. Their Cetus tool provides a compiler infrastructure, which targets C programs, and makes most important parallelization passes by source-to-source transformations. In contrast, we target assembly code level parallelization.

Mathews *et al.* [17] propose a tool that automates the insertion of OpenMP directives to facilitate parallel processing on a wide range of Symmetric Multi Processor (SMP) machines. It statically decomposes a sequential C program into coarse grain tasks, analyzes dependency among tasks and generates OpenMP parallel code. The authors exploit TLP by static analyzes, source-to-source transformations, and code generation, to improve performance beyond the limits of loop parallelism, verified on x86 machine with two test programs. Similarly, we target SMP by a static automatic parallelization method, but on the machine level using more fine-grained ILP.

*2) Dynamic Automatic Optimization Methods*

Yardimci *et al.* [18] propose a method for dynamic automatic performance improvement of optimized sequential binaries. In fact, they have made a software layer for recompilation of a binary which produces parallelized and/or vectorized code by control speculation, loop DOALL parallelism, and parallelization of recursive routines. Their techniques are partly overlapping with ours in that we both exploit loop parallelization, but the loop DOACROSS parallelism in our case. Still, our techniques rely on static scheduling so these two approaches are complementary.

Liu *et al.* [19] describe the POSH, a fully automated Thread-Level Speculation compiler built on the top of gcc. The POSH partitions the code into tasks leveraging subroutines and loops with simple profiling. In contrast to ours, their dynamic tool is only validated on simulation model of multiprocessor chip with four superscalar cores.

Kim *et al.* [20] make the first automatic speculative DOALL (Spec-DOALL) parallelization system for clusters. Spec-DOALL is made from parallelizing compiler and speculative runtime. It minimizes communication and validation overheads of the speculative runtime, gaining the speedup of 43.8x on a 120-core cluster. On the other hand, we target multicores using static ILP of machine code.

Oh *et al.* [21] present techniques for automatic extraction of parallelism within scripts applicable across different dynamic scripting languages. Combining a script with its interpreter, through program specialization techniques, the author's technique embeds any parallelism within the script into the combined program that is extracted by enhanced speculative automatic parallelization techniques afterwards. Evaluated against two open-source script interpreters (Lua and Perl) with 6 input linear algebra kernel scripts each, the technique gains a speedup of 5.10× on a 24-core machine. In contrast to dynamic scripting languages, our technique is used for automatic parallelization of machine code.

*3) Parallelization Methods for FPGA Processor Prototypes*

Wang *et al.* [22] contribute to runtime optimization of sequential programs by proposing MP-Tomasulo. MP-Tomasulo is a task-level out-of-order execution model for sequential programs on FPGA-based multiprocessor. They achieve more than 95% of the theoretical peak speedup for different types of inter-task data dependencies with the prototype system on a real FPGA hardware platform. As opposed to this approach, we propose the parallelization of sequential programs by static compile-time scheduling, verified on a similar FPGA-based multicore processor.

Dou *et al.* [23] make a unified scalable co-processor structure with a linear array of processing elements, implemented on a FPGA chip. It is used as a framework for executing partitions of large-scale matrix decomposition algorithms by a fine-grained pipeline. Static partitioning is limited to four different matrix decomposition algorithms. Similarly, we use a multicore processor model implemented on a FPGA, but our parallelizator partitions any general problem to processor cores.

Dali *et al.* [24] present the parallel architecture with the idea of lower resource utilization compared to other architecture, shown on the example of Fast Fourier Transform (FFT). On the other side, we focus on simple development of C code with automatic parallelization for multicore processor implemented on a FPGA, disregarding possibly higher resource utilization.

*4) Graph-Partitioning Based Optimizations*

The idea of partitioning graphs for parallelization purposes is already described and evaluated in some papers. Capko *et al.* [25] and Capko *et al.* [26] use a similar method for partitioning large data models for further processing. They have represented the model as a DDG. DDG is partitioned using METIS [7] algorithms initially. The partitioning is tuned with several strategies later on.

Hormati *et al.* [27] make a flexible compilation framework named Flextream which dynamically adapts running application to the changing characteristics of the target architecture. They use static compilation like we do in this paper, but with dynamic adaptation techniques at later stages, specifically during execution. StreamIt compiler is used as the starting point and the METIS algorithm for graph partitioning, beside implemented heuristics. Flextream focuses on the area of streaming, whereas we target fine-grained machine instruction-level scheduling for general applications. They have elaborated the solution on a heterogeneous multi-core system (IBM Cell processor), while we target homogenous SMP systems.

## III. Assembly Code Parallelization

As explained in the introduction, the machine code at the binary level, i.e. assembly code, is taken as the input of the parallelizator. Furthermore, the assembly code has already been proved to be suitable for parallelization by Kovacevic *et al.* [12]. It can be produced by compiling a source code (e.g. C program), or even by disassembling executable code. The idea in this approach is to parse the input assembly file, make certain transformations of it, parallelize it by producing the separate assembly code for each core, and then to assemble the executable file for each core.

In order to increase the clarity of method flow and the clearness of the presented details of significant blocks, we will give a simplified description of code intermediate representation (IR) used during its parallelization. IR is a tree data structure consisting of the following elements: (a) the program – the root element that represents the whole assembly program and contains one or more functions, (b) the function – it corresponds to one assembly function, and contains one or more basic blocks, (c) the basic block (BB) – the part of the code that starts with a label and ends with a new label or a instruction, and contains one or more instructions, (d) the instruction – it represents an assembly instruction containing operands, and (e) the operands – leaves that could be shared among several instructions.

The actual flow of parallelization follows. The program written in C code (e.g. file "foo.c") is passed to gcc compiler with compile option "-s", which produces assembly code instead of executable machine code after compile procedure. Assembly code (file "foo.s") is then parsed and translated to IR. The remaining blocks process each BB separately. Each BB undergoes transformations to SSA form, and to DDG which is finally partitioned in DDG partitioner block. BB is then parallelized by scheduling instructions to different cores considering the information from partitioning stage as the starting point. Synchronization instructions are added if necessary. Since new variables are possibly introduced at different stages, linear register allocation is performed on each BB, for each processor core. Finally, the code emitter produces a separate assembly file for each processor core from the parallelized IR. New text segment is made by joining all the BBs for the particular core. Data segment is extended with new memory locations. Executable file is assembled for each core from the matching program code.

There are seven main parts which can be distinguished in this approach: (a) Assembly Code Parser, (b) Register Renaming, (c) DDG Builder, (d) Load-Store Address Analysis, (e) METIS-Based Partitioner, (f) Static Instruction Scheduler, and (g) Register Allocator. The rest of this section presents their detailed description and relations.

### A. Assembly Code Parser

This block is an assembly syntax analyzer that translates textual files with assembly code to programming objects, i.e. previously described data structure called IR. This analyzer reads non-empty lines one by one and tokenizes each line, splitting it to instruction type and operands by a simple state machine. For example the line

```
ADD $t1,$t2,$t3
```

will be separated to four tokens and the instruction will be made from them. *ADD* will be recognized as the instruction type, *$t1* as the destination (defined) operand, whereas *$t2* and *$t3* will be recognized as the source (used) operands.

By parsing each line separately, *Assembly Code Parser* fills IR data structure, i.e. separates it into the program, functions, and BBs. Separation of these is done by recognition of assembler directives and known constructions. Consequently, each instruction has its type, and two lists, namely *usesList* and *defsList*, which are the lists of used and defined variables, respectively.

### B. Register Renaming

In the most cases data dependencies represent a bottleneck for parallelization and other optimization techniques. In order to reduce data dependencies among instructions, redefinition of same variables, i.e. writing to one variable more than once has to be avoided. SSA form meets these requests and, since our approach works on the scope of BB, we can consider the simplified SSA form. Applied to a single BB, the SSA form can be interpreted for our needs as following: each variable is defined (written) exactly once, and it can be only used (read) afterwards [28]. This is achieved by splitting each variable into its different instances. The new instance of the same variable is made as a new variable with the new name built by adding a suffix (unique ID) to the root (the name of the original variable). A new version of the variable is made for each definition. In this way, we make the use-def chains very simple, with one element only. The results of transforming a source code to its SSA form will be given in the example later in the text.

In the case of assembly language for target architectures (e.g. MIPS, Microblaze, etc.), transforming IR to SSA form can be accomplished by register renaming, as registers are the main connection and consequently the main dependency among instructions. Registers that are vital for correct execution of assembly program have to be excluded from transformation to SSA form. Renaming these registers, i.e. variables, and assigning new variables to different registers could lead to inconsistent program execution. The set of these registers is architecture-specific (e.g. assembler temporary register, global pointer register, etc.).

The transition to SSA form of IR is done for each BB, in a single pass, by increasing instance number of each variable at each place of its definition. At the same time, the new variable with the instance number as suffix to the root name is introduced to IR.

### C. DDG Builder

The main problem with parallelization is related to choosing the representation of the sequential code, suitable for partitioning and scheduling to processor cores. During partitioning, the number and the cost of dependencies among partitions have to be minimal. Abstracting the program as a graph, dependence structures that exist in the program are well covered. The DDG derived across all instructions is the structure already used for partitioning [12]. In this case, instructions are represented as vertices of the graph, whereas data dependencies between the two instructions represent edges. In order to determine the dependencies among instructions, the variables from each instruction are observed. Two instructions are considered to

be dependent if they either use or define the same variable.

DDG is built as a list of all instructions, where each instruction has a list of instructions it depends on. The cost of instructions (graph vertices weight) evaluated inside *Assembly Code Parser* block and the cost of dependencies among them (graph edges weight) are needed for partitioning. At the time of building the graph, the weight of each edge is evaluated as the number of variables that two instructions share. The weight of vertices is used by *METIS-based Partitioner* to calculate the load for each partition and make it balanced, while the weight of edges is used to determine the validity of cutting an edge(s) in order to split a partition and make new partitions execute in parallel. In this case, cutting edges will mean adding synchronization instructions for transferring data from one core to another.

### D. Load-Store Address Analysis

According to Bernstein's conditions [29], two program segments $P_i$ and $P_j$ are independent and can run in parallel if and only if (1) is satisfied:

$$\begin{aligned} &I_j \cap O_i = \varnothing, \\ &I_i \cap O_j = \varnothing, \text{ and} \\ &O_i \cap O_j = \varnothing \end{aligned} \tag{1}$$

where $I_i$ are input variables and $O_i$ output variables of $P_i$, and likewise for $P_j$. Applied to the level of instructions, two instructions are dependent if and only if they both occupy at least one same variable, where at least one instruction writes to that variable. In assembly, this can be a problem for both registers and memory. The problem with register variables is neutralized by the previously described register renaming technique. However, the memory problem remains. Hence, simultaneous access to the same memory location, where at least one is a write access, have to be prohibited.

The easiest way to avoid this is to consider memory as one location, and never to schedule write instruction in parallel with any memory access instruction. However, it is also the least efficient way, as it considerably reduces parallelism. Analyzing addresses of memory access (store and load) instructions, some cases can be recognized, and some (but not all) locations accessed by these instructions can be determined as different, as stated in [30]. Therefore, these store and load instructions can be considered as independent and scheduled to run in parallel. Currently, this tool uses a simple, but still efficient method for memory analysis supported by the fact it works with assembly code. Namely, assembly code often addresses memory locations with the same base address, but different offset. It is typically used by store and load instructions, like:

```
lw $t0, 4($t1)
```

where *$t0* is the destination, *$t1* is the base address, and 4 is the offset from *$t1* that makes memory location address completely defined. In this case, the word from (*$t1*+4) address will be read to *$t0*. In this analysis, two load/store instructions addressing different locations are determined with the base address variable having the same value, while the offset value differs. Comparing the values of the base address register variables is not needed since, in described modification of SSA formed IR, a register variable always have the same value starting at the place of its definition. This makes the analysis very simple, yet extremely beneficial, as it proved to discover many independent instructions. This is so because addressing different locations with the same base address and different offset is very frequent, e.g. when addressing elements of an array.

Amme *et al.* [31] make an approach to determination of data dependences in assembly code by a sophisticated algorithm for symbolic value propagation. It derives not only address-based, but value-based dependences between memory operations as well. It is used for optimization of assembly language (e.g. for increasing ILP) producing more precise dependence analysis in many cases and could be used as an optimization technique for our approach as well.

### E. METIS-Based Partitioner

The parallelization method presented in this paper uses METIS Multilevel k-way [7] algorithm for partitioning a graph, where the graph is made from assembly code considering several dependency types, based on available resources on target architecture as the main bottlenecks during the process. All these dependences can be treated as data dependencies, which is the reason why DDG is made as previously described. The information from partitioning DDG is used later as the starting point of parallelization with static instruction scheduling. Specifically, partitioning is refined later, but the first assumed destination of each instruction is the one determined by this block.

Based on DDG, METIS algorithm is used to split instructions (graph) to different processor cores (partitions). The algorithm splits the graph into a given number of partitions, maintaining the load balancing across partitions according to the previously defined costs of instructions (vertices weight). It tries to keep the number of edge-cuts minimal considering edges' weights defined once the DDG is built. An example of splitting DDG is illustrated in Fig. 1. In the given example, the partitioner cuts the edge that has a weight of 2. That is the best solution in the example, as it makes one partition of 3 vertices (instructions) with the total weight of 7, and second one with 4 vertices with the total weight of 8. All other solutions would result in worse load balancing, or in higher weighted edge-cuts. As the edge with the weight of 2 is cut, this means that 2 instructions have to be added (one to each partition and not in parallel). In the worst case this results in a block being 2 instructions longer.

In order to maintain the program semantics some instructions like macroinstructions are prohibited from being

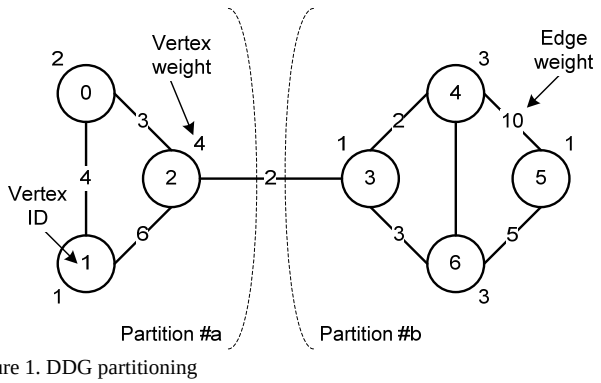


Figure 1. DDG partitioning

mapped to different partitions. For example, the instructions from one macroinstruction have data dependence as they can implicitly use a specific register like temporary register *$acc*. Scheduling these instructions to different cores would result in cutting hidden connections and inconsistent program execution at the end, so they have to remain on the same core.

### *F. Static Instruction Scheduler*

Using the information from partitioning, the static scheduler places instructions into appropriate partition, in order to fit different processor cores. It can also decide to reorder some instructions, so as to achieve better load balancing for processor cores. Finally, it adds prologue instructions at the beginning, and epilogue instructions at the end of each BB of each core. This provides both *execution synchronism* and *data synchronism* to all cores.

*Execution synchronism* enables all the cores to start the following BB from the same point which guaranties data and control consistency. It is similar to Barrier operation used by OpenMP [3], OpenCL [4], Python, etc. As opposed to the mentioned techniques, *execution synchronism* is a generalized barrier operation since the cores reaching the same barrier do not have to run the same code.

*Data synchronism* enables all the cores to start the following BB with the same data set. It is similar to data reduction technique used in TBB [2] as it assumes intra-core communication for data distribution in minimal number of transfers.

Like the *METIS-based Partitioner*, *Static Instruction Scheduler* has to treat some instructions specifically, i.e. to prevent some instructions from being reordered. For example, the instructions from one macroinstruction must not be interleaved with an instruction from the other macroinstruction, i.e. instruction that implicitly uses a specific register like *$acc*.

Another important role of this block is data synchronization across the cores. This is necessary when instruction on one core (*core#a*) uses a variable that is defined in a different core (*core#b*). That is enabled by adding instructions for data synchronization, i.e. by storing a variable to memory from *core#b* after definition, and loading it from memory to *core#a* before usage, but after the definition on *core#b*.

The scheduling algorithm is implemented as follows. For the first step, N core blocks (*CB*) are made, where N is the number of cores. Each *CB* is filled with instructions for the corresponding processor core. Then the algorithm goes through all instruction lists *IL* of the basic block *B*, and for each instruction *doUrgencyRule* routine is called, which in turn separates the list of ready instructions to *W* list. Further on, the *W* list is processed in a loop until it becomes empty, where *doSyncRule* routine is called for each instruction *I* from *W*. As the next step, *doFillRule* routine is called for *W*. First instruction *I* from *W* is finally moved to *CB*.

Once *IL* is processed, each *CB* is made equal length by adding *NOP* instructions. As the last step, instructions from *epilogue* of *B* are added to each *CB*. This is usually a branch or *execution synchronism* instruction along with *data synchronism* instructions as the end of the BB.

The pseudocode of scheduling algorithm is given next:

```
Input: Basic block B, number of cores N.
Output: B partitioned to coreBlocks.
for i = 1 to N do
   CB[i] ← {};
end
IL ← B.instructionList();
while IL ≠ ∅ do
   W ← doUrgencyRule(IL);
   while W ≠ ∅ do
      foreach instruction I ∈ W do
         doSyncRule(I);
      end
      doFillRule(W);
      T ← W.top();
      CB[T.core] ← CB[T.core] ∪ T;
      W ← W \ T;
   end
end
maxLen ← max{C_x | C_x = |CB[x]|, 1 ≤ x ≤ N};
for i = 1 to N do
   len ← |CB[i]|;
   for j = 1 to maxLen – len do
      CB[j] ← CB[j] ∪ NOP;
   end
   CB[i] ← CB[i] ∪ B.epilogue();
end
return CB;
```

Routine *doUrgencyRule(IL)* processes the list of instructions *IL*. It checks if there are ready instructions in *IL*. If there are some ready instructions, it takes all these instructions from *IL* and moves them to the list of ready instructions *RIL*. *RIL* is also the return value of this routine. To consider some instruction as a ready instruction, all registers that it uses have to be already defined or to be input registers (registers that are already defined as well, but in some other block, before the currently processed one). Let us consider the following example:

```
LW $3, 12($4)
LW $2, 0($5)
ADDU $7, $3, $2
ADDI $11, $7, 2
```

Input registers, as inferred from previously generated control flow graph (*CFG*), are $4 and $5, thus instructions that use only these registers can be moved from *IL* to *RIL*. These are the first two (*LW*) instructions. Instructions *ADDU* and *ADDI* stay in *IL* because they use registers that are not defined yet ($2, $3, and $7). In the next call of *doUrgencyRule* routine, *RIL* is empty again and registers $3 and $2 are defined, so *ADDU* instruction becomes ready and can be moved from *IL* to *RIL* and so on, until *IL* becomes empty.

Routine *doSyncRule(I)* takes single instruction *I* that will be scheduled to *core#a*, and checks if it uses a register that is already defined, but on the other core, *core#b*. If yes, this routine adds synchronization instructions (load/store) to the cores. The instruction for storing data from the register to the memory (*SW*) is added to *core#b* after defining the register, while the instruction for loading data from memory to the register (*LW*) is added to *core#a* before the register is used. Two synchronization instructions must not execute simultaneously on different cores. In such a case, *NOP* instruction is added to *core#a*. The example of the code after the synchronization is:

```
# core#0                # core#1
NOP                     LI $3, 4
NOP                     SW $3, 34($0)
LW $3, 34($0)
ADDI $4, $3, 1
```

In the given example instruction *ADDI* from *core#0* uses register \$3 that is defined with instruction *LI* on *core#1*, so the synchronization instruction *SW* is added to *core#1* after instruction *LI*, whereas instruction *LW* is added to *core#0* after *SW*, but before *ADDI* instruction. Furthermore, *NOP* instructions are added to *core#0* in order to wait for definition and storing of register \$3 to memory.

Routine *doFillRule(W)* changes *NOP* instructions from *W* with some useful instructions if possible. *W* is considered as the list of ready instructions, and ready instructions are mutually independent, thus they can be reordered. Removing unnecessary *NOP* instructions is done by reordering instructions in *W*. With such an approach, the number of instructions in *W* can be reduced, which can be noticed from the following example of lists *W* (for 2 cores):

```
# core#0                # core#1
ADDI $2,$0,15           LW $7,4($4)
SW $2,34($0)            NOP
                        LW $11,34($0)
                        ADDI $9,$7,2

# after doFillRule routine
# core#0                # core#1
ADDI $2,$0,15           LW $7,4($4)
SW $2,34($0)            ADDI $9,$7,2
                        LW $11,34($0)
```

In the given example, a *NOP* instruction is removed from *core#1*. Instruction *NOP* is replaced by the instruction *ADDI \$9,\$7,2* by reordering because it is ready, i.e. all the registers it uses are already defined at the place of *NOP* instruction.

### *G. Register Allocator*

By resource allocation we assume the allocation of physical resources from target architecture, specifically processor registers and memory, for the data represented by IR. In this case, we consider register allocation only, since some of the variables connected to registers in sequential code are renamed, i.e. new variables are introduced with transition to SSA form.

Unlike most compilers that use the algorithm of graph coloring for register allocation, this approach uses linear register allocation. Linear register allocation is up to 70 times faster than conventional register allocation [32]. On the other hand, its allocation efficiency is not behind the conventional algorithm for registers allocation in the average case [33]. In order to allocate registers after the transformations made during the transition to SSA form, it is necessary to repeat the liveness analysis. Liveness analysis is done by a single pass through all the instructions. Each *CB* is a slice of sequential code so, consequently, the only necessary information for analysis is the ordinal number of the instruction that uses a variable, in the instruction list. TABLE I shows the results of transforming code to SSA form, liveness analysis, and register allocation. In the given example, the resource *res1* lives in all 4 instructions. The register *R1* is assigned to it. The resource *res2* lives in the instructions 2 and 3 with the register *R2*, while *res3* with the register *R3* lives in the instructions 3 and 4.

TABLE I. SSA ANALYSIS AND REGISTER ALLOCATION FLOW

| Source code | SSA form | Live variables | Allocated regs |
|---|---|---|---|
| A=5 | A.1=5 | A.1=res1 | R1 |
| B=7 | B.1=7 | A.1=res1,<br>B.1=res2 | R1, R2 |
| A=A*B | A.2=A.1*B.1 | A.1=res1,<br>B.1=res2,<br>A.2=res3 | R1, R2, R3 |
| A=A+1 | A.3=A.2+1 | A.2=res3,<br>A.3=res1 | R1, R3 |

Only general purpose registers are used for the register allocation, while other, more specific registers are neither allocated during the allocation, nor renamed during the transition to SSA form. General purpose registers that are available for the allocation are put inside a set structure available in Standard Template Library (STL). The uniqueness of registers is provided using the STL set. The operation of allocating register for a variable that starts to live corresponds to taking the element from the set of unused variables. Once the life of the variable is ended (by the last usage of it), the register assigned to it is returned to the set of unused variables. Spilling of a variable did not happen for any benchmark given later.

## IV. TARGET ARCHITECTURES

The class of supported architecture has certain limitations due to the initial assumptions made within this approach. Hence, there are conditions that the platform to be described has to meet. Due to simple data synchronizations with load-store instructions across processor cores, all the cores have to be connected to a one-level shared memory. Likewise, the architecture must not contain cache memory, as explained and stated before.

Multicore MIPS32 platform, as a SMP with a shared memory meets all these conditions. Register instructions that use 32 general purpose registers have specified execution times. Another benefit is the already existing C compiler (gcc) for producing assembly code. For the first validation step, the software simulation model of MIPS32 platform was used with the simulator from Open Virtual Platforms™ called OVPsim™ simulator. Different numbers of MIPS32 cores along with one shared memory were used during verification. Specifically, OR1K SMP platform was used. A separate file with assembly code was made for each core and assigned to it. During the simulation, the processor model was able to execute parallelized assembly code in the expected number of steps (instructions).

The validation of this approach on hardware was much more challenging. A model of a processor that meets the given requirements is defined and implemented on FPGA for that purpose and used as target architecture later on. Currently the most frequently used soft-processor on FPGA from Xilinx is MicroBlaze™. It is 32-bit RISC Harvard architecture and can be used without cache memory. Further, more than one processor cores can be instantiated, so the model of 1, 2, and 4 cores is implemented in order to verify the scalability of the approach on the examples given below. A block diagram of target architecture with 4 cores is shown in Fig. 2. All the cores are connected to the same shared memory via AXI-4 bus. A similar concept was used by Matheou *et al.* [34] for the validation of their thread

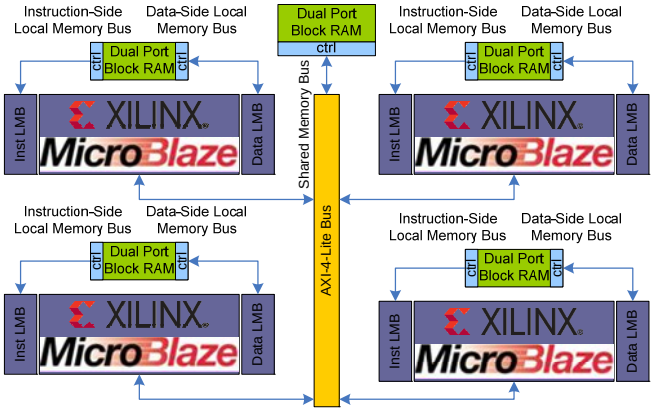


Figure 2. Quad-core Xilinx MicroBlaze™ target architecture

scheduling approach. In the case of connecting multiple cores to the same shared memory, the arbiter within the memory controller (next to Dual Port Block RAM) serves and handles the memory access requests that come from multiple cores. As was already mentioned, each memory access is considered as the access to the same memory location, so the possibility of simultaneous memory access is disabled, and thus also the possibility of nondeterministic instruction execution order that would corrupt the semantics of parallel programs. Besides shared memory, each core is connected to local instruction/data memory via LMB bus. Data segment is mapped to the shared memory so that cores can exchange data. All other segments are mapped to the local data/instruction memory (e.g. text segment, stack, heap, etc.), so that each core can execute its own code independently. The additional advantage of using processor model on FPGA, besides running on hardware, is software simulation. In that way, the approach can be validated even on instruction cycle level. The described processor model was synthesized for XC6SLX150 device from Xilinx Spartan 6 family, with core clock of 108 MHz and resource utilization of 9%, specifically 1% of occupied slice registers and 5% of occupied LUTs.

Definition and implementation of the processor model for manycore architecture is also planned. However, such architecture could be hardly implemented by using one shared memory since the memory arbiter for handling memory access from all the cores would become too complex. A system with more memory levels would be needed and, consequently, a more complex mechanism of data synchronization. Efficient communication between processor cores implemented on FPGA would be demanding. Resource-efficient communication network is given by Wang *et al.* [8].

## V. Experimental Evaluation

During experimental evaluation, different benchmarks were needed to verify this approach, to measure execution speedup, and to show how the approach deals with general problems. Specifically, 4 benchmarks were used: DCT without branches, DCT with branches (DCT WB), Strassen matrix multiplication, and N-Queens problem.

DCT stands for Discrete Cosine Transformation, and for the first benchmark it is implemented without using loops, while for the second one it is implemented with loops (i.e. branches). Branches are expected to introduce a number of smaller BBs, and to reduce the speedup accordingly.

The Strassen algorithm, named after Volker Strassen, is the algorithm for matrix multiplication, faster than a standard matrix multiplication algorithm and useful for large matrices. It has low data dependencies, so the good speedup was expected to be reached after the parallelization.

N-Queens problem solves how to put N queens on the chess board sized N by N. Specifically, the implementation of 8-Queens is used for the last benchmark, as it is a complicated algorithm with large data dependencies.

Experimental evaluation is done by estimating the program execution time, which is then validated on different platform simulation models. Simulation models with 2, 4, 8, and 16 cores were included in the validation process. The performance measure used in the validation process is the speedup, which is defined as follows. Let $T_s$ be the execution time of sequential code, and $T_p$ the execution time of parallelized code. Then the speedup $S$ is defined as:

$$S = T_s / T_p \tag{2}$$

The speedup for each benchmark, for 2, 4, 8 and 16 cores is given in TABLE II. The calculated average speedup from the last column is added to illustrate the average speedup of selected benchmarks. At the very end, the estimated results and the approach itself are verified on the described target architectures with 2 and 4 cores.

The speedups shown in TABLE II for each benchmark are overall speedups (speedups for whole programs). Curves for DCT and DCT WB are also shown in the following figures. In order to underline the speedup dependence on the size of the BB, the speedup curves for both the smallest and the largest BB are also included in following graphs for these benchmarks. This is important because overall speedup is often degraded by a number of small BBs, which are usually not part of the algorithm, but of the initialization or deinitialization of the benchmark. Since the main part of a algorithm (consisted of bigger BBs) is usually included in a loop, it can be easily shown that the total execution time can be approximated to the execution time of bigger BBs for large-enough number of iterations.

The proposed approach gives an overall speedup of 6.68x for 8 cores and even 11.21x for 16 cores for DCT benchmark. However, the speedup of the largest BB is 7.47x for 8 cores, as can be seen in the graph in Fig. 3. This is slightly worse than the speedup achieved by the parallelizator proposed in [11] which is on the other hand capable of parallelizing only a single BB, without support for branches in the code.

For DCT WB benchmark, DCT for blocks or matrices sized 4 by 4 was run. Despite the included loops and branches, BBs were made big enough by optimization used during compile-time. Hence, DCT has large BBs and weak dependencies inside them, giving almost linear speedup

TABLE II. Parallelized Program Execution Speedup

| Cores No | Speedup | | | | Avg Total |
|---|---|---|---|---|---|
| | DCT | DCT WB | Strassen | 8-Queens | |
| 2 | 1.87 | 1.93 | 1.85 | 1.36 | 1.75 |
| 4 | 3.77 | 3.77 | 3.25 | 1.60 | 3.10 |
| 8 | 6.68 | 6.68 | 5.13 | 1.70 | 5.05 |
| 16 | 11.21 | 11.93 | 6.83 | 1.70 | 7.92 |

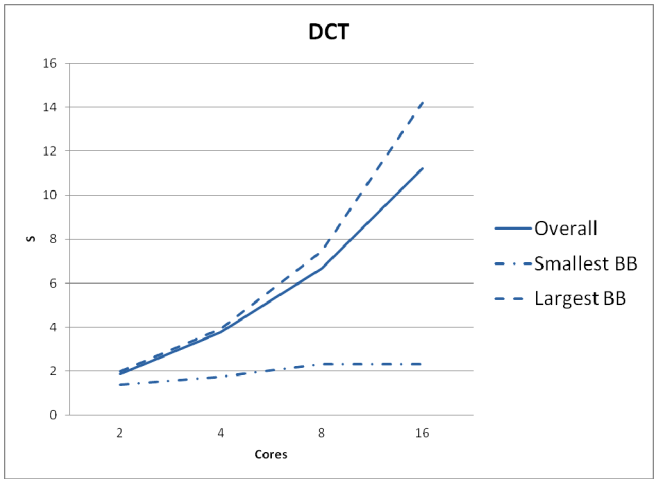


Figure 3. Speedup *S* analysis for DCT benchmark

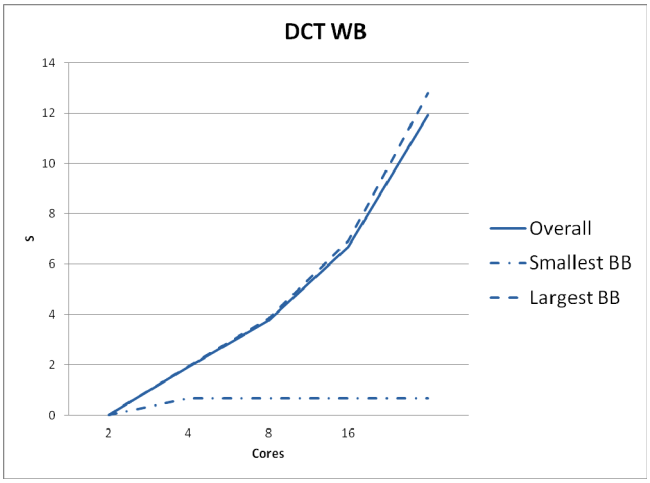


Figure 4. Speedup *S* analysis for DCT WB benchmark

after parallelization as can be noticed in Fig. 4. The achieved speedup is still lower than the one achieved by Vranic *et al.* [11], but that approach and the approach from Kovacevic *et al.* [12] do not support branches and consequently cannot run this benchmark entirely.

The speedup of the largest BB of Strassen matrix multiplication benchmark is almost linear (7.13x for 8 cores, and 12.61x for 16 cores) and that is the consequence of low data dependencies in the multiplication algorithm. Still, a number of small BBs with very low speedup (e.g. the smallest BB has a speedup of 1x) have a significant effect on the overall speedup which is reduced accordingly. Nevertheless, the overall speedup for 8 cores is 5.13x, which is better than the speedup achieved by Vranic *et al.* [11] for the same benchmark.

For 8-Queens benchmark, the growth of speedup by adding processor core is more interesting concerning the largest BB than the overall speedup. The increase is noticeable until 8 cores, where it is around 4.16x. There it reaches its maximum and saturation is noticeable for more than 8 cores. This is the consequence of 8 blocks with low dependences that can be extracted from the BB by the proposed algorithm, and it is caused by the size of board (N=8). Although there are more than 8 cores (e.g. 16 cores), the tool ill create only 8 partitions. More than 8 partitions would render lower speedup due to the large edge-cuts, and a lot of added synchronization instructions. The overall speedup for this benchmark is about 1.7x for 8 cores as a consequence of the larger number of small BBs that directly downgrades parallelism.

The comparison of speedup *S* for all benchmarks and different number of cores is given in Fig. 5. It can be noticed that *S* increases with the increases number of cores (for each benchmark) until the saturation, as well as by lowering the number of branch instructions (e.g. 8-Queens with more branch instructions than DCT, also shows lower speedup).

## VI. Conclusion

Assembly code parallelization is a promising trend [12]. In particular, [12] recognized the method proposed by [25-26] as promising for the assembly code parallelization purpose. Naturally, further research and improvement was needed, but the already proved practice was obeyed.

This paper provides an approach for automatic assembly code parallelization by a refined METIS-based graph partitioning method. It uses static compile-time or just-in-time scheduling of an executable code. Nevertheless, it enables optimal load balancing that gives significant speedup for selected benchmarks. The average speedup of 7.92x is achieved for 16 cores. The speedup is up to 14x for 16 cores in some cases. It is scalable (evaluated using processor performance models and simulation models with 2, 4, 8, and 16 cores), and gives better results for bigger BBs within the benchmark. Besides, the approach is verified on the described target architecture with 2 and 4 cores proving the results estimated by performance model as correct.

The proposed approach can be useful to researchers and developers in the area of compilers and different optimization tools for parallel architectures. Furthermore, it can be utilized either as an idea and basis for further optimizations, as the back-end of a compiler, or as the complete method for code parallelization.

The tool is currently developed in two tracks: (1) upgrading existing optimizing C compiler for DSP platforms [10], and (2) developing a standalone tool for multicore embedded RISC architectures (e.g. MIPS, MicroBlaze, etc.), which could be used as a post-build parallelization tool after compiling a C code (e.g. with gcc). Both tracks consider the application of the given approach in the form of a complete parallelization tool for hard real-time embedded systems.

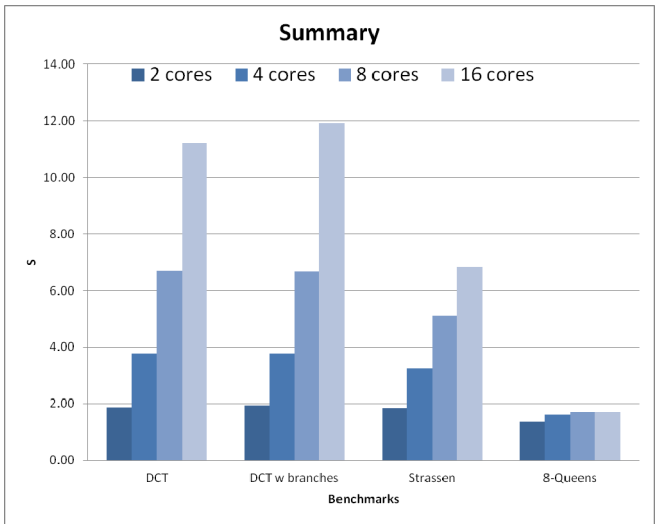


Figure 5. The speedup *S* comparison for all benchmarks

Further research concerning this approach will go in two directions. The first direction is the validation on manycore architecture similar to the one implemented in FPGA with 16 cores or more. The second direction will be the improvement of parallel loops recognition. Currently, we support DOACROSS loop parallelism which is ideal for the loops with loop-carried dependencies. At the same time, the analysis of independent loops similar to the method provided by Kyriakopoulos *et al*. [13] and parallelization based on DOALL loop parallelism are planned as the next step in the improvement of the given tool.

## ACKNOWLEDGMENT

This work has been partly supported by Serbian Ministry of Education and Science of the Republic of Serbia, Grant TR 32031.

## REFERENCES

[1] L. Hochstein, J. Carver, F. Shull, S. Asgari, V. Basili, "Parallel programmer productivity: A case study of novice parallel programmers," Proceedings of the 2005 ACM/IEEE conference on Supercomputing (SC '05), Washington, pp. 35-43, 2005. doi:10.1109/SC.2005.53

[2] M. Popovic, M. Djukic, V. Marinkovic, N. Vranic, "On task tree executor architectures based on Intel parallel building blocks," Computer Science and Information Systems, vol. 10, no. 1, pp. 369-392, 2013. doi:10.1109/ECBS.2012.8

[3] R. Chandra, R. Menon, L. Dagum, D. Kohr, D. Maydan, J. McDonald, "Parallel programming in OpenMP", pp. 157-159, Academic press, 2001, ISBN: 1558606718.

[4] D.B. Kirk, W.W. Hwu, "Programming massively parallel processors", pp. 68-70, Mogran Kaufmann Publishers, 2010, ISBN: 0124159923.

[5] A. Bhattacharjee, G. Contreras, M. Martonosi, "Parallelization Libraries: Characterizing and Reducing Overheads," ACM Trans. Archit. Code Optim, vol. 8, no. 1, pp. 5:1-5:29, 2011. doi:10.1145/1952998.1953003

[6] A. Kotha, K. Anand, M. Smithson, G. Yellareddy, R. Barua, "Automatic Parallelization in a Binary Rewriter," Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 43), Washington, pp. 547-557, 2010. doi:10.1109/MICRO.2010.27

[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, pp. 359-392, 1998. doi:10.1137/S1064827595287997

[8] X. Wang, S. Thota, "A resource-efficient communication architecture for chip multiprocessors on FPGAs," J. Comput. Sci. Technol., vol. 26, no. 3, pp. 434-447, 2011. doi:10.1007/s11390-011-1145-4

[9] U. Vishkin, "Is multicore hardware for general-purpose parallel processing Broken?," Communications of the ACM, vol. 57, no. 4, pp. 35-39, 2014. doi:10.1145/2580945

[10] M. Djukic, M. Popovic, N. Cetic, I. Povazan, "Embedded Processor Oriented Compiler Infrastructure," Advances in Electrical and Computer Engineering, vol. 14, no. 3, pp. 123-130, 2014. doi:10.4316/AECE.2014.03016

[11] N. Vranic, V. Marinkovic, M. Djukic, M. Popovic, "An approach to parallelization of sequential C code," 2011 Second Eastern European Regional Conference on the Engineering of Computer Based Systems, Bratislava, pp. 143-146, 2011. doi:10.1109/ECBS-EERC.2011.30

[12] D. Kovacevic, M. Stanojevic, V. Marinkovic, M. Popovic, "A solution for automatic parallelization of sequential assembly code," Serbian Journal of Electrical Engineering, vol. 10, no. 1, pp. 91-101, 2013. doi:10.2298/SJEE1301091K

[13] K. Kyriakopoulos, K. Psarris, "Non-linear symbolic analysis for advanced program parallelization," IEEE Transactions on Parallel and Distributed Systems, vol. 20, no. 5, pp. 623-640, 2009. doi:10.1109/TPDS.2008.131

[14] G. Ottoni, R. Rangan, A. Stoler, D. I. August, "Automatic thread extraction with decoupled software pipelining," Proceedings of the 38th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 38), Washington, pp. 105-118, 2005. doi:10.1109/MICRO.2005.13

[15] S. Campanoni, T. Jones, G. Holloway, V. J. Reddi, G. Y. Wei, D. M. Brooks, "HELIX: Automatic Parallelization of Irregular Programs for Chip Multiprocessing," Proceedings of the Tenth International Symposium on Code Generation and Optimization (CGO '12), San Jose, pp. 84-93, 2012. doi:10.1145/2259016.2259028

[16] C. Dave, H. Bae, S. Min, S. Lee, R. Eligenmann, S. Midkiff, "Cetus: A source-to-source compiler infrastructure for multicores," Computer, vol. 42, no. 12, pp. 36-42, 2009. doi:10.1109/MC.2009.385

[17] M. Mathews , J. P. Abraham, "Automatic Code Parallelization with OpenMP task constructs," Proceedings of the 2016 International Conference on Information Science (ICIS '16), Kochi, pp. 233-238, 2016. doi:10.1109/INFOSCI.2016.7845333

[18] E. Yardimci, M. Franz, "Dynamic parallelization and mapping of binary executables on hierarchical platforms," Proceedings of the 3rd Conference on Computing Frontiers (CF '06), Ischia, pp. 127-138, 2006. doi:10.1145/1128022.1128040

[19] W. Liu, J. Tuck, L. Ceze, W. Ahn, K. Strauss, J. Renau, J. Torrellas, "POSH: a TLS compiler that exploits program structure," Proceedings of the Eleventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '06), New York, pp. 158-167, 2006. doi:10.1145/1122971.1122997

[20] H. Kim, N. P. Johnson, J. W. Lee, S. A. Mahlke, D. I. August, "Automatic speculative DOALL for clusters," Proceedings of the Tenth International Symposium on Code Generation and Optimization (CGO '12), San Jose, pp. 94-103, 2012. doi:10.1145/2259016.2259029

[21] T. Oh, S. R. Beard, N. P. Johnson, S. Popovych, D. I. August, "A Generalized Framework for Automatic Scripting Language Parallelization," Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques (PACT '17), Portland, pp. 356-369, 2017. doi:10.1109/PACT.2017.28

[22] C. Wang, X. Li, J. Zhang, X. Zhou, X. Nie, "MP-Tomasulo: A Dependency-Aware Automatic Parallel Execution Engine for Sequential Programs," ACM Trans. Archit. Code Optim, vol. 10, no. 2, pp. 9:1-9:26, 2013. doi:10.1145/2459316.2459320

[23] Y. Dou, J. Zhou, G.-M. Wu, J.-F. Jiang, Y.-W. Lei, S.-C. Ni, "A unified co-processor architecture for matrix decomposition," J. Comput. Sci. Technol., vol. 25, no. 4, pp. 874-885, 2010. doi:10.1007/s11390-010-9372-7

[24] M. Dali, A. Guessoum, R. M. Gibson, A. Amira, N. Ramzan, "Efficient FPGA Implementation of High-Throughput Mixed Radix Multipath Delay Commutator FFT Processor for MIMO-OFDM, " Advances in Electrical and Computer Engineering, vol.17, no.1, pp. 27-38, 2017. doi:10.4316/AECE.2017.01005

[25] D. Capko, A. Erdeljan, G. Svenda, M. Popovic, "Dynamic repartitioning of large data model in distribution management systems," Electronics and Electrical Engineering, vol. 120, no. 4, pp. 83-88, 2012. doi:10.5755/j01.eee.120.4.1461

[26] 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, pp. 41-46, 2011. doi:10.4316/AECE.2011.04007

[27] A. H. Hormati, Y. Choi, M. Kudlur, R. Rabbah, T. Mudge, S. Mahlke, "Flextream: Adaptive compilation of streaming applications for heterogeneous architectures," The 18th Int. Conf. on Parallel Arch. and Compilation Techn., Washington, pp. 214-223, 2009. doi:10.1109/PACT.2009.39

[28] A. V. Aho, M. S. Lam, R. Sethi, J. D. Ullman, "Compilers: principles, techniques, & tools", pp. 369-370, Addison-Wesley, 2007, ISBN: 0321486811.

[29] A.J. Bernstein, "Analysis of programs for parallel processing," IEEE Transactions on Electronic Computers, vol. EC-15, no. 5, pp 757-763, 1966. doi:10.1109/PGEC.1966.264565

[30] S. Debray, R. Muth, M. Weippert, "Alias analysis of executable code," Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '98), San Diego, pp. 12-24, 1998. doi:10.1145/268946.268948

[31] W. Amme, P. Braun, F. Thomasset, E. Zehendner, "Data dependence analysis of assembly code," Int. J. Parallel Program., vol. 28, no. 5, pp. 431-467, 2000. doi:10.1023/A:1007588710878

[32] C. Wimmer, M. Franz, "Linear scan register allocation on SSA Form," Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO '10), Toronto, pp. 170-179, 2010. doi:10.1145/1772954.1772979

[33] M. Puletto, V. Sarkar, "Linear Scan Register Allocation," ACM Trans. Program. Lang. Syst., vol. 21, no. 5, pp. 895- 913, 1999. doi:10.1145/330249.330250

[34] G. Matheou, P. Evripidou, "Verilog-based simulation of hardware support for data-flow concurrency on multicore systems," Proceedings of the 2013 International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation, pp. 280-287, Samos, 2013. doi:10.1109/SAMOS.2013.6621136