MIPT: Rapid Exploration and Evaluation for Migrating Sequential Algorithms to Multiprocessing Systems with Multi-Port Memories

Gorker Alp Malazgirt and Arda Yurdakul
Department of Computer Engineering
Bogazici University
Bebek, Istanbul, Turkey
alp.malazgirt, yurdakul@boun.edu.tr

Smail Niar
LAMIH
University of Valenciennes
Valenciennes, France
sniar@univ-valenciennes.edu

Work In Progress

***Abstract*—Research has shown that the memory load/store instructions consume an important part in execution time and energy consumption. Extracting available parallelism at different granularity has been an important approach for designing next generation highly parallel systems. In this work, we present MIPT, an architecture exploration framework that leverages instruction parallelism of memory and ALU operations from a sequential algorithm's execution trace. MIPT heuristics recommend memory port sizes and issue slot sizes for memory and ALU operations. Its custom simulator simulates and evaluates the recommended parallel version of the execution trace for measuring performance improvements versus dual port memory. MIPT's architecture exploration criteria is to improve performance by utilizing systems with multi-port memories and multi-issue ALUs. There exists design exploration tools such as Multi2Sim and Trimaran. These simulators offer customization of multi-port memory architectures but designers' initial starting points are usually unclear. Thus, MIPT can suggest initial starting point for customization in those design exploration systems. In addition, given same application with two different implementations, it is possible to compare their execution time by the MIPT simulator.**

***Keywords*—Instruction Level Parallelism; Genetic Algorithm; Optimization; Measurements**

I. Introduction

Today, there is significant research for developing new platforms for large scale concurrent and distributed systems. Most of the research aims towards increasing scalability by raising the abstraction levels and providing simplicity to develop and maintain code. Hence, the single computing block (node, actor etc.) performance of these large scale systems has remained inefficient due to poor system level design decisions and/or unoptimized code [1]. In addition, different platforms have different compute node characteristics, such as shared threads or dedicated single thread per node.

Building more powerful compute units has been suggested in [2]–[4] for increasing scalability and reducing node inefficiency problems in future large scale processing systems. A recent publication [5] suggests more aggressive use of instruction extensible processors at compute nodes of large scale processing systems in order to tailor application for improved performance and efficiency. Authors target three different parts of the processor architecture: the instruction decoders, register files and the data paths. For memory intensive applications, authors consider reduction of load/store operations as major contributors to performance improvements. They have shown that overall performance have improved by reduction of data load/store by using multi-port register file. In addition, short lived data was stored in custom buffers instead of register files which prevent unnecessary register file accesses. This short lived data also decrease space in the register file for non-intermediate data and causes additional memory load/stores, thus performance is decreased. However, it is known that increasing port numbers affects area dramatically which affects power consumption negatively [6], [7]. This issue has become more important with the advent of processor architectures with multi-port memories [8], [9].

In this paper, we present the architecture and underlying heuristics of MIPT (Million Memory Instructions Per Trace) framework which has built its foundations from the aforementioned discussion. It processes sequential algorithm execution traces for architecture exploration and extracts parallelism for memory and Arithmetic Logic Unit (ALU) operations. MIPT heuristics explores the design space and decides the number of read/write ports for memory operations and the number of issue slots for ALU operations. In addition, it recommends a new schedule that exploits the suggested port and issue slot number. MIPT aims to improve instruction level parallelism (ILP) by first analyzing memory load/store instruction traces and true register data dependencies[a] between memory instruc-

[a]True register data dependency is defined as dependency between registers due to data flow, reading a value of register after writing it.

tions. Our analysis algorithm extracts maximum and average instruction level parallelism. Multi-port memory number and issue slot sizes are determined by maximum parallelism. Our recommendation algorithm explores design space and reduces maximum port usage and issue slot size. Hence, we gain significantly from resource usage while keeping the same performance. This is achieved by reordering some instructions to different access steps, thus lowering maximum parallelism. Nevertheless, we try to preserve average parallelism, which is defined as the average number of instructions per access step.

Our primary contributions are:

- A rapid analysis method that extracts memory level parallelism of memory instructions and instruction level parallelism of ALU operations. The method is based on bundling load/store instructions together without creating any data hazards. This lowers the number of accesses to memory and exploits multi-port memory structures. After memory operations are bundled, arithmetic operations are bundled and scheduled in the same way.
- An evolutionary algorithm that reduces maximum memory port usage which gains significantly from resource usage but keeping the similar degree of parallelism and performance. This is achieved by re-scheduling some instructions, thus improving average parallelism without increasing the execution time.
- A custom simulator that evaluates the performance of the recommended architecture. In addition, MIPT simulates and reports the execution time given algorithm's execution trace, read/write port and ALU slots size.

The main practical application and usefulness of MIPT framework is providing a starting point for migrating sequential applications to multiprocessing systems with multi-port memories. Given a real sequential application, MIPT finds out how it can be parallelized. MIPT framework does not implement any hardware architecture. MIPT analyzes any sequential application, finds average and maximum parallelism, searches the design space and recommends designer memory port and issue slot size. There exists design exploration tools such as Multi2Sim and Trimaran [10], [11]. These simulators offer customization of multi-port memory architectures but require a huge execution time for exploring the different configurations. Thus, our tool can suggest initial starting point for customization in those design exploration systems. In addition, given same application with two different implementations, it is possible to compare their execution time by our custom simulator. It is known that coding style, compiler optimizations and different compilers generate different binaries. Conversely, given an algorithm, its input data, read/write port size and multi-issue ALU structure, it can simulate and suggest a schedule which minimizes execution time

The rest of this paper is organized as follows. We discuss related work in literature and a use case where a framework like ours could be used in Section II. Section III explains the architecture and the stages of the framework. Experimental results are presented in Section IV. Section V includes our discussion of potential MIPT use cases and our conclusion.

II. Related Work

We have identified three areas where parallelism is exploited, thus relate to our work:

Compilation Techniques Software compilation techniques based on traces and profiles have been studied widely [12] [13] [14] [15] [16]. Parallelizing compilers for VLIW processors provide ILP by formatting more than one operations as one instruction at compile time. VLIW compiler prevents implementing dependency checking, structural hazard checking and complex instruction dispatch hardware. All these are figured out by the compiler. Software pipelining, trace scheduling, predicated execution, hierarchical reduction and speculative execution have been major compiler optimizations. Trace scheduling [12] required additional code when operations are reordered. Speculative execution [13] helps the reduction of compensation code and moves instructions past branches. The speculatively executed code shouldn't produce any stalls to the processor pipeline when it is not needed. Speculative execution can be implemented as hardware or software. Software pipelining [14] [16] aims at compacting loop kernels by minimizing initiation intervals. Predicated execution [15] aims to convert branches to basic blocks with hardware defined predicates on certain instructions. The predicated execution simplifies compilation due to reducing branch code and avoid branch prediction but predicated instruction are always fetched whether or not they are executed. Hierarchical reduction [14] is the method to simplify scheduling process by compacting and representing scheduled program components as a single component. This component is consisted of having the aggregate resource and scheduling constraints of the scheduled program components. Approaches that extract ILP parallelism that do not work with execution traces require dependency analysis methods such as Omega test [17], GCD test [18] and points-to-analysis [19] which are computationally expensive. MIPT heuristics analyze execution traces where all the address and register dependencies can be checked via real memory addresses and registers generated in the traces by only iterating over the control and data flow graph (CDFG). Therefore, all of the data dependency and structural hazards can be solved allowing code movement above branches if branch is known to be not taken under provided input data. This analysis is explained in the next section.

Parallelism extraction and classification: Parallelism extraction/classification from applications are used in application specific processor design and measurements of software metrics [20] [21]. Work in [20] presents methods for measuring available instruction level and thread level parallelism in different classes [22] of applications. Results show that applications which are in the same class do not possess same ILP and TLP parallelism. In addition, TLP and ILP do not necessarily correlate. Applications with low ILP might have

high TLP. However, thread level analysis and task partitioning depends highly on the underlying communication structures. Our framework leverages instruction level parallelism from sequential algorithms. It does not support any thread level parallelism extraction. Work in [21] introduces force based parallelism on basic blocks to measure VLIW processor issue width. Force based parallelism calculates possible availability of operations in a time interval inside basic blocks by keeping distribution graphs. Inter iteration dependence of loops are broken with introduction of local variables. However, handling of branches with distribution graphs are known to generate poor results, because some paths can be favored probabilistically [12]. Conversely, MIPT processes execution traces and no paths are favored. All existing paths in the given trace are processed by our heuristics. Nevertheless, if a possible branch path is not executed, it is not processed.

Cache replacement Policy and Scratchpad memory utilization: Parallelization of memory instructions is a determining criteria when designing new cache replacement policies. The work in [23] proposes a DRAM cache replacement policy in out of order superscalar processors based on parallelism cost and occurrence time of parallel memory operations. Additionally, policy switch is possible based on hardware based sampling. Similarly Data Trace Cache [24] inserts an application specific cache with data locality improving cache placement function. This improves hit rates in tree based applications because conflict sets are allocated based on tree levels. So memory is divided accordingly based on level cardinality. Similarly, parallelism of memory instructions is exploited for efficient allocation and utilization of scratchpad memories (SPM) where it coexists with caches. The heuristics developed in [25] the required scalar variable data size, access frequencies and execution time of block sizes for maximizing scratchpad utilization. Similarly, our recommendation algorithm can be used in allocation and utilization of SPM in embedded systems domain. Based on the available port requirements, our recommendation algorithm tries to extract parallelism while lowering the number of accesses.

III. Framework Architecture

MIPT framework is divided into four stages as shown in Figure 1. First step is profiling of the algorithm. Application execution traces and data dependency graph are created from given input data set and program binary. It is explained in Section III-A. Second step of MIPT is the analysis of traces. In the analysis section, analysis algorithm applies certain rules to traces and the trace is reordered according to the rules. Thus, a new reorder of traces, maximum parallelism and average parallelism are generated. These results are fed to recommendation algorithm which is the third step. The details of the analysis algorithm is explained in Section III-B. The recommendation algorithm aims to reduce maximum memory port and issue slot size usage without decreasing average parallelism.

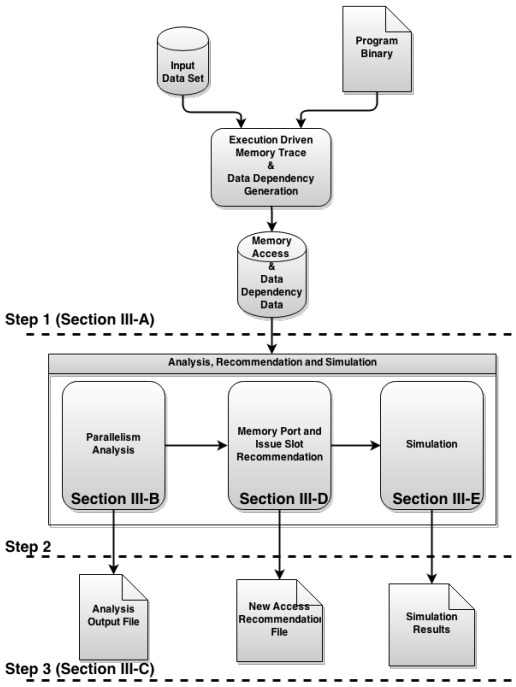

Figure 1: MIPT process flow

Recommendation algorithm generates a new trace from the analyzed trace. Generated multi-port multi-issue schedules are simulated by MIPT's custom simulator, which is explained in Section III-E. Details of the recommendation algorithm is explained in Section III-D. MIPT simulator generates reporting metrics. Reporting metrics can be used for application exploration and for comparing different applications. Section III-C explains details the metrics.

A. Extracting and Formatting ExecutionTraces

Traces are generated from the application binary and given input data set. Our trace generation program uses PIN [26] library and x86 application binaries are compiled from C/C++ source files. The traces are stored in MIPT's local database.

B. Parallelism Analysis from Execution Traces

Our analysis algorithm processes traces to extract parallelism. To extract maximum parallelism, at the beginning, we assume that there are a sufficient number of internal registers. Register renaming is employed to prevent structural hazards. The analysis is concerned with reordering and bundling instructions without breaking data dependencies. Hence, the reordering logic is developed under certain rules.

- A set of n instructions $M = \{I_1, I_2.....,I_n\}$ with execution times $\{p_1, p_2, ..., p_n\}$
- For $I_i, I_j \in M$, $A_s \in \mathbb{N}$, $k \in \mathbb{N}$, and $t \in \{read, write\}$; the quadruple $\langle I_i, A_s, t, k\rangle$ denotes the kth occurrence (access step) of I_i at address A_s which is a read or a write instruction
- For all $k > 0$ such that:

1) $\langle I_i, A_s, write, k+1\rangle$ cannot start before or at the same time with $\langle I_j, A_s, write, k\rangle$
2) $\langle I_i, A_s, read, k+1\rangle$ cannot start before or at the same time with $\langle I_j, A_s, write, k\rangle$
3) $\langle I_i, A_s, write, k+1\rangle$ cannot start before or at the same time with $\langle I_j, A_s, read, k\rangle$

The algorithm applies these rules to each load/store instruction. The instructions that satisfies these rules exploit instruction parallelism without breaking data dependencies. Rule 1, 2 and 3 are concerned with data hazards that may happen when write and read to same address occur by different instructions. The original order read from the trace files is not changed in order to preserve data dependency. Rule 3 also handles register data dependency. All the rules are checked in a single pass of memory instructions. We define the reordering problem as the reordering of instructions based on the given rules and finding their execution start time.

Definition 1. *A reordering α of all the given instructions M and $\forall k > 0$;*
$S^{\alpha}_{\langle I_i,A,t,k\rangle}$ is the start time of $\langle I_i, A, t, k\rangle$ based on given reordering rules.

Instructions with the same start times are parallel and are scheduled at the same access time. This reordering extracts the maximum parallelism which is defined as the highest number of instructions in an access step. The algorithm 1 shows how the reordering logic rules are applied to traces.

Input: Execution trace list and data dependency graph
Output: Reordered trace list, maximum parallelism and average parallelism metric
initialize reordered trace list;
initialize access step pointer;
foreach *instruction in the execution trace list* **do**
  check reordering rules against instructions which access step pointer points to in reordered trace list;
  if *all the reordering rules are satisfied* **then**
    add instruction to reordered trace list at current access step;
  else
    increment access step pointer;
    add instruction to reordered trace list at the new access step;
  end
end

Algorithm 1: Analysis algorithm which applies the reordering rules to instructions and reorders a trace

The input of the algorithm are execution trace list and data dependency graph. The algorithm iteratively compares instructions and bundles them to access steps according to the reordering logic. The output of the algorithm is the reordered trace list, average parallelism and maximum parallelism obtained from the trace.

C. MIPT Outputs

1) Reporting Metrics: MIPT prepares the following metrics: *total number of registers used, memory port size, ALU issue size, ALU issue structure, average parallelism, maximum parallelism, estimated execution cycle, estimated average number of memory accesses (number of bundles), estimated average non-memory accesses (number of ALU bundles)*. These information yield explorations and comparisons between different algorithms or different versions of the same algorithms.

2) Recommended Schedule Generation: MIPT produces recommended schedule for the explored configuration. The recommended schedule is the new reordered trace file. The trace file can be explored through MIPT's user interface.

D. An Evolutionary Algorithm for Reducing Maximum Parallelism of Instructions

Maximum parallelism information extracted from analysis algorithm is important because fastest execution requires maximum parallelism. However, implementations based on maximum parallelism will be inefficient in terms of area and power consumption. The recommendation algorithm reduces maximum port usage and issue slot size. Hence, we gain significantly from resource usage while keeping the performance attained at maximum parallelism. This is achieved by reordering some instructions to different access steps, thus lowering maximum parallelism. The algorithm is based on a genetic algorithm (GA) in the literature [27]. We have selected this method because the access step and instruction encoding could be converted to GA chromosome encoding without much effort. In addition, GA can solve multi solution problems because of its population concept.

Recommendation algorithm extracts instruction level parallelism. We have chosen to limit the issue size to the maximum number of read/write port number of the multi-port memory. For example, 3R 1W configuration is assumed to have less than or equal to four issue slots. Overall execution time can be improved by employing different parallelisms such as thread level parallelism or multi-core parallelism in addition to instruction level paralellism. Moreover, future 3D memory chips such as work in [28] are multi-port, supports 4R 3W configuration and are faster than current dual port DRAM memories.

Solution Encoding: Solution encoding is designed as one dimensional array. Each element of the array represents an instruction. This array is called an individual and each instruction is a gene of the chromosome. For example, given a chromosome (1,3,2,2), it is understood that there are four instructions and first instruction executes at access step one, second instruction is at step three and the rest is at step two. A solution is feasible if it does not violate any dependency rules explained in Section III-B.

Fitness Function: The fitness of a chromosome is identical

to the objective function value of the solution. The objective is to minimize the maximum number of instructions in an access step without increasing the total access steps and creating any data hazards. All the reordering has to satisfy data dependencies, otherwise the solution is considered infeasible.

Population Size: There is a trade-off in evolutionary algorithm design. Large population sizes are slower than small population sizes. However, small population sizes may not generate enough diversity. In order to balance speed and diversity we limit an instruction to be reordered in a predetermined number of access steps. This is called access step window (ASW). Let n being number of selected access steps which have instructions more than initial average parallelism and a being the highest number of access step window (ASW). ASW is determined from dependency graph by checking inter iteration dependencies in the loops if there are any. ASW is bounded to 10 as the maximum value. Finally, The population size is calculated as:
$p(n,p) = max\left\{2, \lceil \frac{ln(n*a)}{a} \rceil\right\} * 10 * log(n)$

Initialization of the Population and Selecting Parents: New orders and parent selection are generated randomly until the population size is filled. Infeasible, solutions are not kept in the population pool.

Generating New Members: A new member is generated by crossover method from two parents that are chosen randomly from the pool. The crossover happens from a randomly chosen point of the genes. Nevertheless, the crossover point should not violate data dependency. For example, two parents that are selected from the pool are (1, 3, 2, 5, 4, 6) and (2, 2, 3, 6, 5, 4). If crossover index is three, offsprings become (1, 3, 2, 6, 5, 4) and (2, 2, 3, 5, 4, 6). The data dependency is checked by using the reordering rules presented in Section III-B.

Mutation: We have opted out to apply mutation because of no performance improvements.

Replacement: We admit a solution into the population, if it is distinct and its fitness values is better than the worst fitness value in the population. The worst members is then discarded. This improves the average fitness value of the population gradually while maintaining genetic diversity.

Termination The algorithm terminates after observing $10 * log(n)\sqrt{a}$ successive iterations where the minimum of the maximum parallelism of the solutions has not changed.

The algorithm 2 shows the steps of the procedure. The steps of the algorithm applies the aforementioned features of GA.

E. Simulation of Generated Schedules

Generated schedules are simulated in order to estimate the improvement in performance with calculated port and issue slot size configurations. In our experiments for this work, we have assumed that all the data resides in the cache. We have obtained the operation costs from [29] and [30].

Input: Analyzed reordered trace list, data dependency graph, maximum parallelism and average parallelism
Output: Recommended trace list, recommended port and issue slot size

```
choose initial population;
evaluate each chromosome's fitness;
repeat
    select two random chromosomes from population;
    apply crossover operator;
    foreach created new chromosome do
        if chromosome is a feasible reordering then
            evaluate chromosome's fitness;
            if population replacement criteria is met then
                admit chromosome to population pool;
            end
        end
    end
until terminating condition is met;
```

Algorithm 2: Recommendation algorithm which generates a recommended schedule, lowers maximum parallelism while preserving average parallelism

IV. Experimental Results

We developed a custom simulator in order to simulate reordered traces. As mentioned in Section III-A, we use PIN API [26] for trace generation and CACTI [31] for memory area estimations. We have experimented our approach with string matching algorithms that are shown in Table I. Multi-port multi-issue configurations are compared to dual port single issue configuration which is taken as the baseline configuration. The algorithms are modified from [32].

Input set contains one centimorgan DNA base pairs which is approximately one million characters of text, and ten thousand base pairs for patterns. The text alphabet size is four. Analysis, recommendation algorithms and our custom simulator have been implemented in Java and the results have been analyzed using Octave. String matching algorithms are separated into three classes based on the methods for searching strings. These classes are bit parallelism, automata, and comparison. Although, there exists different types of string matching algorithms, all of them are highly memory intensive as shown in Figure 2, thus data oriented. Control flow of string matching algorithms are similar to sparse matrix multiplication which are typical HPC applications [33], [34]. FSBNDMQ, FAOSO and TVSBS have different implementations. In FSBNDMQ, we change the q grams and the selected lookahead values. For example, FSBNDMQ21 means the algorithm works with 2 grams and 1 lookahead value. In FAOSO, alignment numbers vary. In TVSBS, we used different window sizes.

Our hardware setup is Intel 2.9 GHz Core i7 8GB DDR3 running 64-bit Unix OS. CACTI [31] memory area estimations use following parameters: Block size: 64 bytes, Size: 1 GB,

Abbreviation	Algorithm	Type
FSBNDMQ	Forward Simplified Backward Nondeterministic DAWG Matching with q-grams	bit-parallel
BMH-SBNDM	Backward Nondeterministic DAWG Matching with Horspool Shift	bit-parallel
KBNDM	Factorized Backward Nondeterministic DAWG Matching	bit-parallel
FAOSO	Fast Average Optimal Shift Or	bit-parallel
SEBOM	Simplified Extended Backward Oracle Matching	automata
FBOM	Forward Backward Oracle Matching	automata
SFBOM	Simplified Forward Backward Oracle Matching	automata
TVSBS	TVSBS: A Fast Exact Pattern Matching Algorithm for Biological Sequences	comparison
FJS	Franek Jennings Smyth String Matching	comparison
GRASPM	Genomic Rapid Algorithm for String Pattern Matching	comparison

TABLE I: String Matching Algorithms and abbreviations

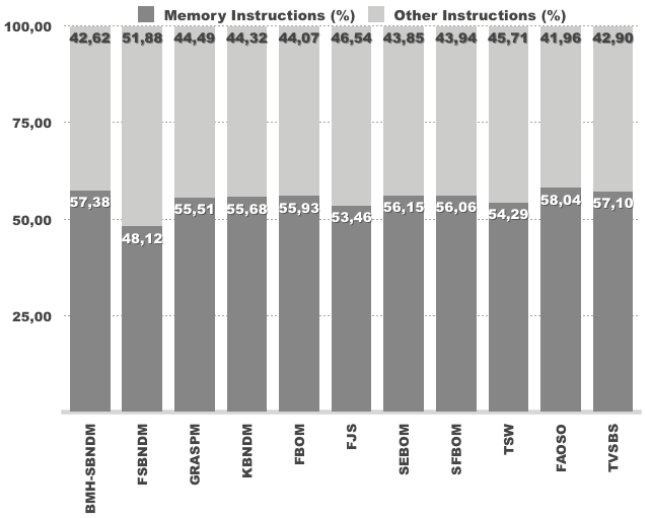

Figure 2: Memory instructions profile of string matching algorithms

technology: 32 nm, page size: 8192 bits, burst length: 8, internal prefetch width: 8, input/output bus width: 64, operating temperature: 350 K and no selected optimizations. All benchmarks have been compiled with GCC 4.8.2 applying -O1 optimizations. We have opted for -O1 optimizations because our experiments have shown that compilation with -O2 has been equivalent to compilation with -O1. Function inlining heuristics that are mostly used in -O2 has not made any improvements. Similarly, compilation with -O3 has not been not consistent. Each time we compile with -O3, we obtain a different binary because some of heuristics are not deterministic. For reproducibility of experimental work, we did not prefer -O3. With given input set, one benchmark consumes approximately one hour of processor time.

Experiments have shown that approaches which aim to maximize parallelism by populating memory ports may not yield best memory configuration, because increasing number of port sizes impact memory area greatly as shown in Analyzed Maximum Parallelism column in Table II. Our recommendation algorithm has managed to reduce area up to 599%. Maximum parallelism extracted from SEBOM requires 22R and 1W configuration. Recommendation algorithm reduces this configuration to 3R and 1W without increasing average parallelism. This reduction in port configuration causes 599% reduction in required memory area.

There are significant differences in execution times between different types of algorithms as shown in Table II. Bit-parallel algorithms have fewer access steps than other types. More than half of the instructions of all the algorithms are consisted of memory instructions. However, there is not any relation between access steps and memory instruction profiles. FSBNDMQ algorithm has the least average parallelism and least ratio of average memory instructions among all algorithms. FAOSO6 has the most maximum recommended parallelism and TVSBS-W8 has the most average parallelism. However, they both execute longer than FSBNDMQ.

The recommended average parallelism does not seem to correlate with execution times. Recommended average parallelism value of FJS algorithm is 4.13 whereas FSBNDMQ variations have 2.85 on average. Nevertheless, FJS has three times more execution time than FSBNDMQ variations. Similarly, Recommended maximum parallelism does not convey any hint about the execution times. GRASPM has lower recommended maximum parallelism than FAOSO6, but it requires more access steps to complete. Algorithm types do not have similar analyzed maximum parallelism. Analyzed maximum parallelism of FSBNDMQ algorithm is read port dominated whereas FAOSO algorithm is dominated by write ports. Variations of the same algorithms can also have different number of analyzed maximum parallelism. TVSBS-W4 has 22 read ports whereas TVSBS-W8 has 6 read ports.

All of the algorithms have shown execution time speed up between 268% to 439 % in multi-port memory over dual-port memory shown in Table II. Parallelizing only memory instructions has produced speed ups. However, the maximum speed up is obtained when both memory operations and ALU operations are parallelized. In Table II, we report the best execution times obtained after memory and ALU operations.

The types of the string matching algorithms also affect parallelism and MP area consumption. Leveraging parallelism of multi-port memory in automata based algorithms causes 1.51x increase in area over dual port memory configuration, however the speed up obtained is 387% on average. Similarly, our recommendation algorithm reduces memory area over our analysis results between 213% and 599% shown in Table II. For example, analyzed maximum parallelism of TVSBS-W4 with given input data set yields 22R, 1W configuration. Our recommendation algorithms lowers the port size to 5R, 1W configuration and this reduction decreases memory area

Benchmark	Analyzed Maximum Parallelism	Recommended Maximum Parallelism	Recommended Average Parallelism	MP Execution time (1000x cycles)	DP Execution time (1000x cycles)	Area Increase MP over DP(times)	Execution Speed up MP over DP (%)	Area reduction of recommendation algoritm after analysis algoritm (%)
BMH-SBNDM	7R, 1W	4R, 1W	3.25	3900	7669	2,20	329	208
FAOSO2	8R, 1W	3R, 1W	3.30	7400	13949	1,51	318	366
FAOSO4	5R, 11W	2R, 6W	3.29	4897	8847	3,40	304	323
FAOSO6	5R, 11W	2R, 6W	4.1	10875	17145	3,40	268	323
FBOM	11R, 1W	3R, 1W	3.54	9932	20946	1,51	354	599
FJS	7R, 2W	3R, 2W	4.13	11489	30258	2,95	439	275
FSBNDM20	6R, 1W	3R, 1W	2.99	1755	3497	1,51	325	244
FSBNDM21	5R, 1W	2R, 1W	2.82	1817	3487	1,36	313	213
FSBNDM31	6R, 1W	3R, 1W	2.82	1704	3195	1,51	308	244
FSBNDM32	6R, 1W	3R, 1W	2.79	2023	3792	1,51	306	244
FSBNDM41	5R, 1W	2R, 1W	2.81	1909	3509	1,36	302	213
FSBNDM42	6R, 1W	3R, 1W	2.80	1915	3517	1,51	302	244
FSBNDM43	5R, 1W	2R, 1W	2.83	1931	3570	1,36	304	213
FSBNDM61	6R, 1W	3R, 1W	2.84	2327	4202	1,51	298	244
FSBNDM62	6R, 1W	4R, 1W	2.88	2326	4227	1,51	300	244
FSBNDM64	6R, 1W	3R, 1W	2.84	2351	4240	1,51	298	244
FSBNDM81	6R, 1W	3R, 1W	2.88	2714	4846	1,51	297	244
FSBNDM82	6R, 1W	3R, 1W	2.87	2712	4835	1,51	296	244
FSBNDM84	6R,1W	3R, 1W	2.86	2711	4814	1,51	295	244
FSBNDM86	6R, 1W	3R, 1W	2.85	3016	5357	1,51	295	244
GRASPM	11R, 1W	3R, 1W	3.70	6445	13800	1,51	359	599
KBNDM	7R, 2W	4R, 1W	3.64	6669	15126	2,20	380	244
SEBOM	11R, 1W	3R, 1W	3.54	9813	20734	1,51	355	599
SFBOM	9R, 1W	3R, 1W	3.48	9986	20925	1,51	352	439
TSW	7R, 1W	4R, 1W	3.26	6395	13874	1,51	363	208
TVSBS-W2	11R, 1W	4R, 1W	3.30	7525	15290	2,20	343	413
TVSBS-W4	22R, 1W	5R, 1W	3.39	8342	17397	2,91	352	526
TVSBS-W8	6R, 7W	3R, 2W	4.16	21847	51775	1,95	387	420
qsort_iter	12R, 1W	4R, 1W	3.35	682	-	1.91	261	295
qsort_rec	3R, 4W	1R, 3W	2.95	710	-	0.28	217	233

TABLE II: Analyzed and recommended multi-port configurations, improvement in execution time with gains in area and area improvement introduced with recommendation algorithm

by 526%. Furthermore, fewer port sizes do not increase the number of access steps. Thus, the average parallelism stays the same. These improvements in area and port size reduction also affect other parameters such as power consumption, energy consumption and delay positively. Nevertheless, handling multiple accesses and ALU operations increase control and arithmetic logic. Hence, aforementioned parameters can be affected negatively. We will investigate the trade-off as the future work. While leveraging ILP, we have aimed at simplicity when designing our algorithms. Because, for accurate extraction of parallelism, the input data size usually tend to be large.

Given one million string characters, one benchmark consumes two hours of processor time in given hardware setup. Recommendation algorithm consumes one hour of processor execution time. The execution time depends on the number of instructions. Execution time increases when the number of instructions increase.

V. Discussion and Conclusion

Given a sequential algorithm, MIPT can be used for rapid exploration of parallelism. All the string matching algorithms we present are sequential. Our analysis and recommendation algorithms have leveraged parallelism and shown improvements when its inherent parallelism is exploited. This is also true with any sequential algorithm given to MIPT. Given an algorithm, MIPT can provide a starting point for migrating software to hardware for hardware co-acceleration through leveraging parallelism. After extracting parallelism,

data path could be designed based on memory and core operations. Pipeline depth and units like register file or cache size could be determined. MIPT can also help to differentiate implementation differences between algorithms. For example, Table II shows recursive and iterative versions of sorting ten thousand elements with Quicksort [35]. The difference between execution times and average parallelism lies in different parameters as well as the well known difference in the nature of recursion and iteration programming paradigms. Pivot point selection can change the number of accesses for both iterative and recursive version. The recursion version needs to have recursive function calls and this can be costly when number of values to sort decrease. In the iterative version, the usage of extra space for swapping values can also increase the number of memory accesses. The effects of these difference design decisions and programming choices can be identified with MIPT framework.In conclusion, we have presented MIPT framework that aims to improve ILP by analyzing execution traces and register data dependencies and recommend a new multi-port multi-issue schedule.

ACKNOWLEDGMENT

This work has been supported by Bogazici University Scientific Research Projects (BAP) Project No: 11A01P6. We also acknowledge support of TUBITAK given to Prof. S. NIAR under Project No: 2221.

REFERENCES

[1] D. Yang, X. Zhong, D. Yan, F. Dai, X. Yin, C. Lian, Z. Zhu, W. Jiang, and G. Wu, "Nativetask: A hadoop compatible framework for high performance," in *Big Data, 2013 IEEE International Conference on*, 2013.

[2] A. Pavlo, E. Paulson, A. Rasin, D. J. Abadi, D. J. DeWitt, S. Madden, and M. Stonebraker, "A comparison of approaches to large-scale data analysis," in *Proceedings of the 35th SIGMOD international conference on Management of data*. ACM, 2009.

[3] IBM. (2013) Big data handbook. IBM. [Online]. Available: http://www-01.ibm.com

[4] A. Verma, N. Zea, B. Cho, I. Gupta, and R. Campbell, "Breaking the mapreduce stage barrier," in *Cluster Computing (CLUSTER), 2010 IEEE International Conference on*, 2010.

[5] R. Hameed, W. Qadeer, M. Wachs, O. Azizi, A. Solomatnikov, B. C. Lee, S. Richardson, C. Kozyrakis, and M. Horowitz, "Understanding sources of inefficiency in general-purpose chips," in *In ISCA 10 Proceedings of the 37th Annual International SymposiumonComputerArchitecture*.

[6] Y. Tatsumi and H.-J. Mattausch, "Fast quadratic increase of multiport-storage-cell area with port number," *Electronics Letters*, vol. 35, no. 25, pp. 2185–2187, 1999.

[7] W. Zuo, W. Zuo, and L. Jiaxing, "An intelligent multi-port memory," in *Intelligent Information Technology Application Workshops, 2008. IITAW '08. International Symposium on*, 2008, pp. 251–254.

[8] *Alpha Architecture Handbook*, Digital Equipment Corporation, Maynard, MA, 1994.

[9] T. Chen, R. Raghavan, J. N. Dale, and E. Iwata, "Cell broadband engine architecture and its first implementationa performance view," *IBM Journal of Research and Development*, vol. 51, no. 5, 2007.

[10] R. Ubal, J. Sahuquillo, S. Petit, and P. López, "Multi2sim: A simulation framework to evaluate multicore-multithread processors," in *IEEE 19th International Symposium on Computer Architecture and High Performance computing, page (s)*, 2007, pp. 62–68.

[11] L. N. Chakrapani, J. Gyllenhaal, W. H. Wen-mei, S. A. Mahlke, K. V. Palem, and R. M. Rabbah, "Trimaran: An infrastructure for research in instruction-level parallelism," in *Languages and Compilers for High Performance Computing*. Springer, 2005, pp. 32–41.

[12] J. A. Fisher, "Trace scheduling: A technique for global microcode compaction," *IEEE Transactions on Computers*, vol. 30, no. 7, pp. 478–490, 1981.

[13] M. D. Smith, M. S. Lam, and M. A. Horowitz, *Boosting beyond static scheduling in a superscalar processor*. ACM, 1990, vol. 18, no. 3a.

[14] M. Lam, "Software pipelining: An effective scheduling technique for vliw machines," in *ACM Sigplan Notices*, vol. 23, no. 7. ACM, 1988, pp. 318–328.

[15] P. Hsu and E. S. Davidson, *Highly concurrent scalar processing*. IEEE Computer Society Press, 1986, vol. 14, no. 2.

[16] V. H. Allan, R. B. Jones, R. M. Lee, and S. J. Allan, "Software pipelining," *ACM Computing Surveys (CSUR)*, vol. 27, no. 3, pp. 367–432, 1995.

[17] W. Pugh, "The omega test: a fast and practical integer programming algorithm for dependence analysis," in *Proceedings of the 1991 ACM/IEEE conference on Supercomputing*. ACM, 1991, pp. 4–13.

[18] D. E. Maydan, J. L. Hennessy, and M. S. Lam, "Efficient and exact data dependence analysis," in *ACM SIGPLAN Notices*, vol. 26, no. 6. ACM, 1991, pp. 1–14.

[19] R. P. Wilson and M. S. Lam, *Efficient context-sensitive pointer analysis for C programs*. ACM, 1995, vol. 30, no. 6.

[20] V. Caparros Cabezas and P. Stanley Marbell, "Parallelism and data movement characterization of contemporary application classes," in *Proceedings of the 23rd ACM Symposium on Parallelism in Algorithms and Architectures*, ser. SPAA 11, 2011.

[21] R. Jordans, R. Corvino, L. Jozwiak, and H. Corporaal, "Exploring processor parallelism: Estimation methods and optimization strategies," in *Design and Diagnostics of Electronic Circuits Systems (DDECS), 2013 IEEE 16th International Symposium on*, 2013.

[22] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams *et al.*, "The landscape of parallel computing research: A view from berkeley," Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Tech. Rep., 2006.

[23] M. K. Qureshi, D. N. Lynch, O. Mutlu, and Y. N. Patt, "A case for mlp-aware cache replacement," in *Proceedings of the 33rd Annual International Symposium on Computer Architecture*, ser. ISCA '06, 2006.

[24] S. Ramaswamy, J. Sreeram, S. Yalamanchili, and K. V. Palem, "Data trace cache: An application specific cache architecture," *SIGARCH Comput. Archit. News*, vol. 34, no. 1, Sep. 2005.

[25] H. Salamy and J. Ramanujam, "An effective solution to task scheduling and memory partitioning for multiprocessor system-on-chip," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 31, no. 5, 2012.

[26] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: building customized program analysis tools with dynamic instrumentation," in *Acm Sigplan Notices*, vol. 40, no. 6. ACM, 2005.

[27] O. Alp, E. Erkut, and Z. Drezner, "An efficient genetic algorithm for the p-median problem," *Annals of Operations Research*, vol. 122, no. 1-4, 2003.

[28] S. K. Lim, "3d-maps: 3d massively parallel processor with stacked memory," in *Design for High Performance, Low Power, and Reliable 3D Integrated Circuits*. Springer, 2013, pp. 537–560.

[29] Intel, *Intel 64 and IA-32 Architectures Software Developer Manuals*. [Online]. Available: http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html

[30] A. Fog. Instruction tables. [Online]. Available: http://www.agner.org/optimize/instructiontables.pdf

[31] P. Shivakumar and N. P. Jouppi, "Cacti 3.0: An integrated cache timing, power, and area model," Technical Report, Compaq Computer Corporation, Tech. Rep., 2001.

[32] S. Faro. (2014, January) Smart string matching algorithms research tool. [Online]. Available: http://www.dmi.unict.it/ faro/smart/

[33] R. Yuster and U. Zwick, "Fast sparse matrix multiplication," *ACM Transactions on Algorithms (TALG)*, vol. 1, no. 1, pp. 2–13, 2005.

[34] F. G. Gustavson, "Two fast algorithms for sparse matrices: Multiplication and permuted transposition," *ACM Transactions on Mathematical Software (TOMS)*, vol. 4, no. 3, pp. 250–269, 1978.

[35] C. A. Hoare, "Quicksort," *The Computer Journal*, vol. 5, no. 1, 1962.