THREAD-LEVEL AUTOMATIC PARALLELIZATION IN THE ELBRUS OPTIMIZING COMPILER

L. Mukhanov
Software department
MCST
Moscow,Russia
email: mukhanov@mcst.ru

P. Ilyin
Software department
MCST
Moscow,Russia
email: ilpv@mcst.ru

A. Ermolitsky
Software department
MCST
Moscow,Russia
email: era@mcst.ru

A. Grabezhnoy
Software department
MCST
Moscow,Russia
email: grab_av@mcst.ru

S. Shlykov
Software department
MCST
Moscow,Russia
email: shlykov@mcst.ru

A. Breger
Software department
MCST
Moscow,Russia
email: breger@mcst.ru

ABSTRACT
Most microprocessor manufacturers are currently moving toward multi-core and multiprocessing systems, because it is becoming very difficult to increase the performance of a single-core microprocessor. To enjoy the advantages of multi-core and multiprocessing systems in applications, software engineers must redesign these applications with appropriate tools. For some applications, such redesign could be very complex and thus expensive. As a result, most microprocessor manufacturers are trying to develop thread-level automatic parallelization functionality in their optimizing compilers. Such compilers allow application developers to enjoy all the advantages of multi-core and multiprocessing systems without redesigning their applications.

In this paper, we discuss automatic parallelization in the Elbrus (EPIC architecture) optimizing compiler. We describe thread-level automatic parallelization for EPIC architectures in detail, as well as the automatic parallelization techniques we developed and the results of our tests.

KEY WORDS
Automatic parallelization,TLP,EPIC,optimizing compiler

1 Introduction

It is very difficult to improve the performance of a single-core microprocessor any further. Typical ways to improve such microprocessors' performance are to increase their logic speed or clock frequency. Increasing the length of the instruction word increases the logic speed (as in VLIW and EPIC architectures). Such an increase allows the microprocessor to execute more instructions per tick (Instruction Level Parallelism, ILP). As the instruction word lengthens, the microprocessor becomes more complex. As the processor becomes more complex, its clock frequency decreases. Decreasing the clock speed may decrease performance of the microprocessor on some applications. On the other hand, new manufacturing technologies and design improvements can increase clock frequencies. In practice, it takes a great deal of effort to achieve these improvements. Because of these challenges, most manufacturers are aiming for multi-core and multiprocessing systems. Multi-core and multiprocessing systems can use Thread Level Parallelism (TLP). Using TLP requires an application to be built with corresponding tools for thread level parallelization. The problem is that many applications are written for uniprocessor systems. Some applications would be complex and difficult to redesign. Because of this difficulty, the problem of thread-level automatic parallelization in optimizing compilers is actual.

2 Elbrus architecture

In this paper, we discuss automatic parallelization in the optimizing compiler for the Elbrus architecture. We describe thread-level automatic parallelization for EPIC architectures, develop automatic parallelization techniques, and give the results of our tests. Elbrus [1] is a 64-bit EPIC (Explicitly Parallel Instruction Computing) architecture with the structure shown in Figure 1. In this architecture, an optimizing compiler schedules instructions [2]. In other words, the optimizing compiler translates all source code instructions into 64-byte instruction words (Very Long Instruction Word), which the microprocessor executes in scheduled order (In-Order execution). The microprocessor has six arithmetic logic units (ALUs), which are distributed between two clusters. Each cluster has its own L1 data cache (D$L1) and its own copy of the register file (RF). Copies of register files are synchronized. The predicate logic unit (PLU) makes it possible to eliminate some control operations from code as it becomes predicated. The Array Access Unit (AAU) helps prevent microprocessor stalls due to L2 data cache misses, using asynchronous data prefetch from memory to a special buffer (Array Prefetch Buffer, APB). Table 1 describes the microprocessor's main technical characteristics. The Elbrus 3M1

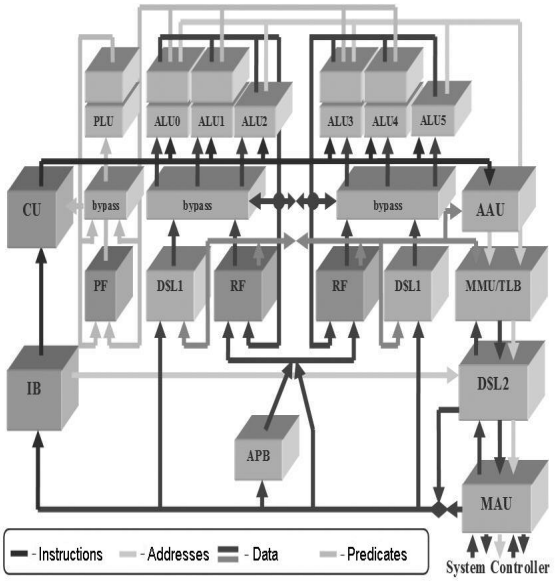

Figure 1. The architecture of the Elbrus microprocessor

computer is built on the base of the Elbrus microprocessor. This computer consists of two Elbrus microprocessors with symmetric shared memory. Detailed information about the microprocessor and the Elbrus 3M1 computer can be found in [3].

3 Automatic parallelization functionality in the Elbrus optimizing compiler

Elbrus is an EPIC architecture with in-order execution. The Elbrus optimizing compiler does most optimizations and schedules most operations. The Elbrus compiler consists of more than 200 optimizations, which make it possible to parallelize programs on the ILP (Instruction-Level Parallelism)[4]. To parallelize programs on the TLP , we developed the specific optimization. This optimization is one phase of the optimizing compiler, and thus can use the same analytical results that other compiler optimization use: for example, the results of pointer analysis, loop dependence analysis, dataflow analysis and control flow analysis [5]. The optimization uses these analysis results not only to find code segments that it can parallelize, but also to transform code into a form suitable for parallelization, as in other compiler optimizations.

We designed the thread-level automatic parallelization phase of the Elbrus optimizing compiler to parallelize the hot loops of a program, which are determined by profiling. In the optimization pipeline, we placed the automatic parallelization phase after the main loop optimizations, most of which make loops more suitable for parallelization. The parallelization phase analyzes all loops. If some loop is suitable for a certain parallelization technique, then the compiler marks the loop. After that, the compiler parallelizes the outermost marked loops using a corresponding technique.

Table 1. The main characteristic of Elbrus microprocessor

Characteristic	
CMOS technology	0,13 micron
Clock frequency	300 Mhz
Quantity of transistors	75 millions
L1 instruction cache	64 KBytes
L1 data cache	64 KBytes
L2 data cache	256 KBytes
peak performance:	
- 64-bit operations	$6,9 * 10^6$ MIPS/ $2.4 * 10^9$ FLOPS
- 32-bit operations	$9,5 * 10^6$ MIPS/ $4,8 * 10^9$ FLOPS
- 16-bit operations	$12,2 * 10^9$ MIPS
- 8-bit operations	$22,6 * 10^9$ MIPS
- cache throughput	8,4 Gb/sec
- memory throughput	4,8 Gb/sec

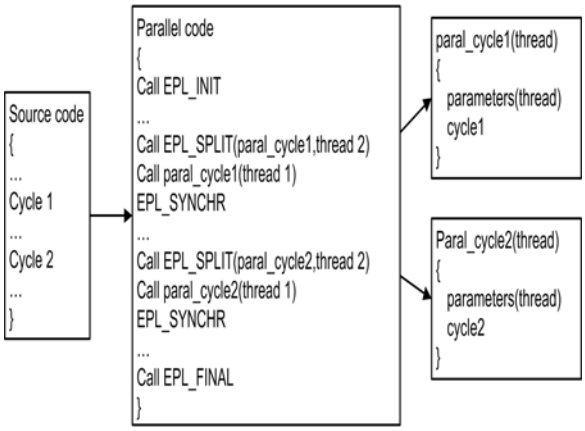

Figure 2. Cutting loops and parallelization of a source code

3.1 Parallelization

A parallelized program initially spawns two threads. Each of these threads is executed on a different microprocessor. During parallelization, the compiler splits marked loops into separate procedures (Figure 2). A loop runs in parallel as a collection of different threads that call the new procedures with different parameters (Figure 3).

In these pictures, EPL_INIT, EPL_SPLIT, EPL_SEND_SYNCHR, EPL_SYNCHR, EPL_FINAL are functions from the parallelization support library (Elbrus Parallel Library). In EPL_INIT (EPL_FINAL) the second thread is created (is stopped) by sending a signal. Initially, the second thread awaits a corresponding signal (EPL_RECEIVE_EVENT) from the first thread that the second thread can execute a procedure with a parallel loop. The thread signals with semaphores

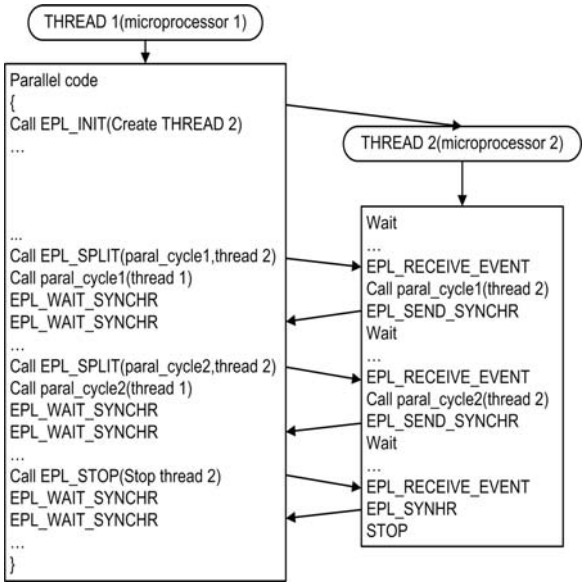

Figure 3. Execution of a parallel code

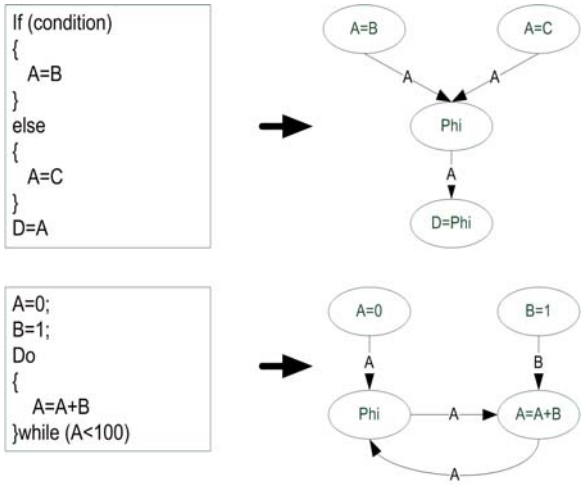

Figure 4. A Def-Use graph

(function EPL_SPLIT). The first thread passes a pointer to the parallel procedure in shared memory to the second thread. The thread's ordinal number, given as a parameter, determines the parameters of a parallel loop in the parallel procedure. After the parallelized loop completes, the first thread awaits the end of the parallelized loop execution on the second thread (EPL_WAIT_SYNCHR). The second thread informs the first thread of the parallelized loop's completion by calling the function EPL_SYNCHR. The first thread can call the EPL_STOP function to stop the second thread.

The compiler splits parallel loops into procedures after loop optimizations and before the main optimizations, which exploit features of the EPIC architecture (for example, software pipelining based on rotating registers[6], data prefetching based on APB, and if-conversion [7]). Thus, the intermediate representation of a program in the optimizing compiler after loop optimizations does not contain operations dependent on the EPIC architecture. This architecture-independence simplifies program analysis and automatic parallelization transformations. Splitting parallel loops into separate procedures makes the intermediate representation simpler by removing the need to represent special parallel instructions explicitly. Adding parallel instructions into the intermediate language would require redesigning all compiler optimizations to work correctly with these instructions, including the instruction scheduling phase (like in the Intel optimizing compiler [8]). These changes could worsen the performance of the optimizing compiler. The effectiveness of optimizations is especially crucial on an In-Order EPIC architecture, because the performance of applications depends on the optimizing compiler: the microprocessor provides no dynamic support (Out-of-Order execution). The Elbrus optimizing compiler can apply different optimization sequences to a given procedure. The compiler chooses the optimization sequence according to certain characteristic parameters of the procedure. Splitting parallel loops into separate procedures makes it possible to optimize these procedures independently from the sequential portions of the program. Scheduling parallel code independently makes better use of microprocessor resources.

3.2 The basic parallelization technique

Determining whether a loop can be parallelized requires several analyses [9]:

1. Control-flow analysis;
2. Dataflow analysis;
3. Pointer dependence and loop dependence analysis.

Control-flow analysis builds a control flow graph (CFG) for the loop. The loop is suitable for parallelization if:

1. The loop is reducible. A reducible loop has only two edges that enter it (including the back CFG-edge of the loop);
2. The loop has one back edge and one exit edge.

Dataflow analysis builds a Def-Use graph (DU-graph). A Def-Use graph is a directed graph (based on SSA [10]), in which nodes either correspond to operations or are phi-nodes. A phi-node combines multiple versions of a variable. Examples of Def-Use graph is given in Figure 4 (on the left side of the figure source codes are given, on the right side - built graphs).
A loop is suitable for parallelization if:

1. The loop has no DU out-edges [5];

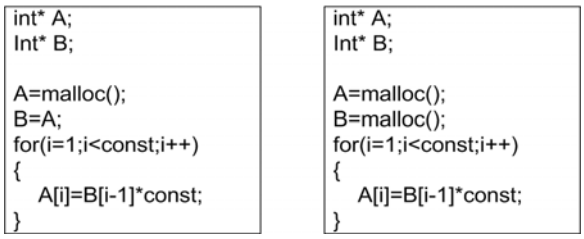

Figure 5. An example of pointer dependence between arrays

2. The loop has only one induction: the basic induction of the loop. The analysis determines the number of inductions from phi-nodes in the loop head. The basic induction must have the form:

$$arg1 = oper(arg1, arg2) \quad (1)$$

In this expression $arg1$ is an object of the induct variable phi-node, $arg2$ is a constant within a loop to be parallelized, $oper$ is a function: add, sub, multiplication, logical or, etc. The function must be reducible;
The basic technique parallelizes a loop on the basic induction. If the compiler has selected the loop for parallelization, then it calculates the up and down bounds for each thread according to the following rules:
$Thread1$:

$$new\_down\_bound = down\_bound;$$

$$new\_up\_bound = (up\_bound - down\_bound)/2 + \\ + down\_bound; \quad (2)$$

$Thread2$:

$$new\_down\_bound = (up\_bound - down\_bound)/2 + \\ + down\_bound + 1;$$

$$new\_up\_bound = up\_bound; \quad (3)$$

These bounds are passed on the stack, and each thread determines its bounds in a cut procedure before a parallel loop begins to execute (the function *parameters* in Figure 2). These bounds have to be constant within the loop. During parallelization, DU edges entering the loop must be broken. The values from these edges are also passed on the stack. Since a loop is parallelized by its basic induction, some operations may access arrays or memory on different iterations. Such operations may impede parallelization. Therefore, the compiler does pointer and loop dependence analysis in the third phase [11]. The first goal of the analysis to determine which arrays are dependent (pointer dependence analysis). For example, the left side of Figure 5 shows dependent arrays A and B. On the right side of the figure, these arrays are independent. In the optimizing compiler, each array is represented by an object in the intermediate representation. Information about pointer dependence relationships between objects is required for loop dependence analysis, which exposes dependencies between operations that access the same objects on different iterations of a parallel loop.

Assume that operations S and T access the same object obj in a loop with t basic variable (i_1). The purpose of loop dependence analysis is to find out when these operations access the same element of the object. The analysis result contributes to determining whether a loop is parallelizable. The left side of Figure 5 shows an example of cross iteration dependence between the same object. Finding these dependencies for each operation means building a ps-form. The ps-form is a polynomial of the following form:

$$p_0 = c_0 + c_{11} \times x_{11} \times x_{12} ... x_{1k_1} + ... + c_{n1} \times x_{n1} \times x_{n2} ... x_{nk_1} \quad (4)$$

In this expression, $c_0, c_{11}, c_{12}..., c_{n1}$ are constants and x_{ij} are variables, corresponding to nodes in the Def-Use graph. Thus, the ps - form is based on operations and phi-nodes. Each element of the object can be represented by an index. If a nest has depth k with basic variables $(i_1, i_2, ..., i_k)$ (where i_1 is the basic variable of the outermost loop and i_k is the basic variable of the innermost loop), then the access by an operation to an element is a function of $(i_1, i_2, ..., i_k)$:

$$index = function(i_1, i_2, ..., i_k) \quad (5)$$

If an index of an element can be represented in the ps-form, then loop dependence analysis proceeds. Otherwise, the analysis recognizes an operation that accesses this element as dependent on other operations in loop dependence analysis. Thus, only indices in the linear form are analyzed. For example, the following index could not be analyzed:

$$index = i^2 + j \quad (6)$$

Two operations are recognized as dependent if for one of these operations an index could not be analyzed. When analyzing loop dependence between two different operations that access elements of the same object, the compiler estimates the subtraction of the previously-built ps-forms for these operations. If the subtraction is in the ps-form, then the compiler builds a system of equations. Into this system, the compiler adds inequalities based on the ps-forms built for up and down loop bounds of the basic induction variable (it also adds equality for a step of the basic induction variable). The compiler builds and adds these inequalities for each loop of a nest. To find out whether there is an exact dependence two operations, the compiler also has to add specific inequalities. For example, it adds such equalities when it requires a dependence in an exact direction [12]. In this context, direction means that one of the analyzed operations occurs earlier than another operation. The optimizing compiler solves such systems with the simplex method [13], [14]. Subtractions are built only from constants and the basic variables. If this difference contains other variables, then the ps-form is confluent and operations are recognized as dependent.

```
for(i=0;i<const;i++)
{
   a[i]=b[i];
}
```

Figure 6. An example of loop in canonical form

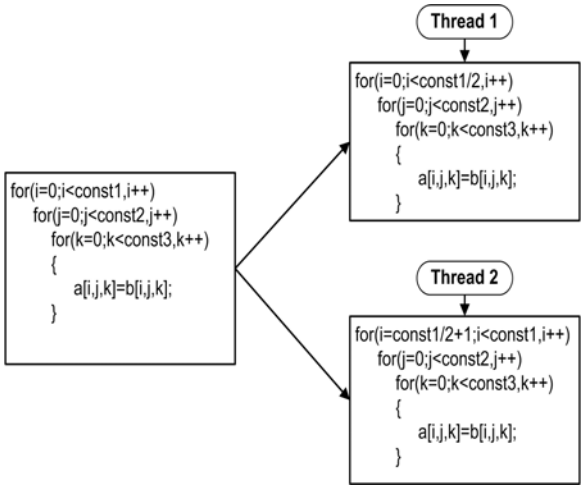

Figure 7. An example of a parallelized loop nest

To apply the basic parallelization technique, there must be no observable read and write operations that access the same memory address on different loop iterations. To apply this technique, a dependence should be unobservable for several write operations.

Let $S(i_1)$ be the index of an element in the object or address in memory, which is accessed by the operations S. $T(i'_1)$ is the index of some other element in the same object or address in memory, which is accessed by the operation T. Then, the following basic technique applies to a loop:

$$\forall S(i_1), \forall T(i'_1) : S(i_1) = T(i'_1) \Leftrightarrow i_1 = i'_1 \qquad (7)$$

In this expression, S is a read operation or a write operation and T is a write operation from a loop to be parallelized (S and T can be the same write operation).

The basic technique parallelizes loops that contain pure calls. A call is pure if it does not write to memory or mutate global variables.

If a loop can be parallelized using the basic technique, then the loop is in canonical form. For example, the loop in Figure 6 is in canonical form. The basic technique applies to outer loops and loop nests. An outer loop is in canonical form if each inner loop of this outer loop is in canonical form. A nest of loops is in canonical form if each outer loop of this nest has only one successor loop (in the loop tree). An outer loop is parallelized by the basic variable of this loop (Figure 7).

In parallel outer loops, we have not observed dependencies between read-write and write-write pairs of operations on different iterations. That is, if an outer loop has a nested loop of depth k with basic variables $(i_1, i_2, ..., i_k)$, then it is possible to parallelize this loop when:

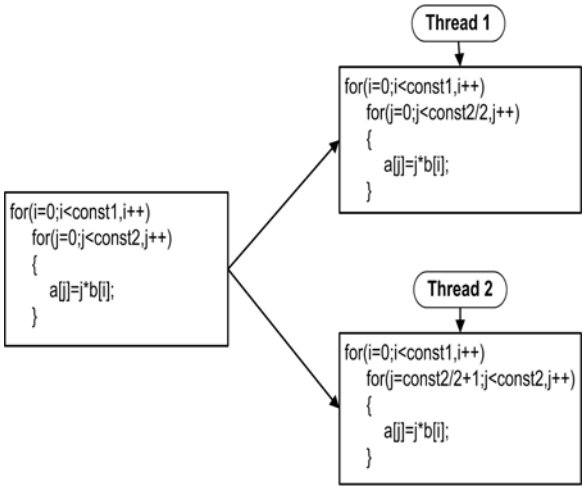

Figure 8. An example of a loop in which only the innermost loop can be parallelized using the basic technique

$$\forall S(i_1, ..., i_k), \forall T(i'_1, ..., i'_k) : S(i_1 ..., i_k) = T(i'_1 ..., i'_k) \Leftrightarrow$$

$$\Leftrightarrow i_1 = i'_1 \qquad (8)$$

In this expression, S is a read operation or a write operation and T is a write operation from a loop to be parallelized (S and T can be the same write operation).

Up or down loop bound transferred to the parallel procedure (passed on the stack) could be a parameter. This is why for correct parallelization dynamic checking have to be embedded to a parallel code. This checking makes possible to find out how many iterations will be performed for a parallelized loop. If the number of iterations is less than two then serial version of a source code is executed.

3.3 Advanced cutting loop technique

The left side of Figure 8 gives a loop in which only the innermost loop could be parallelized using the technique we described. The problem is that synchronization for the parallelized inner loop in the outer loop could make parallelization less effective. As a result, parallelizing such loops would have no positive effect.

This is why we extended our technique to such loops. The main idea of this extension is splitting on the basic variable of the innermost loop as before, but for cutting the outermost loop is used (Figure 8). This extension applies with the basic technique if outer loops are not suitable for parallelization because of loop dependencies. The technique is intended to cut the outermost loops to reduce the influence of synchronization.

Let the outermost loop have depth k where the loop's basic variables are $(i_1, i_2, ..., i_k)$ (i_1 is the basic variable of

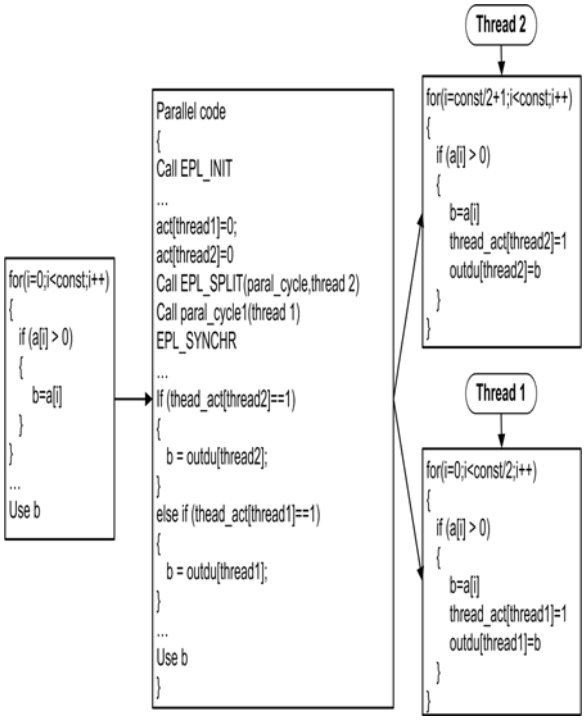

Figure 9. An example of loop with an out DU-edge

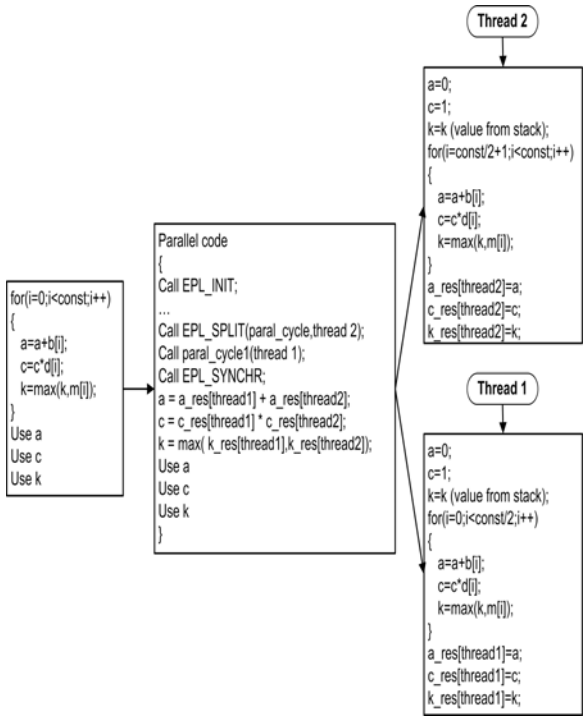

Figure 10. An example of a loop with several inductions

the outermost loop) and the loop with basic variable i_k can be parallelized (but i_{k-1} cann't be parallelized). Then the outermost loop can be cut if:

$$\forall S(i_1,...,i_k), \forall T(i'_1,...,i'_k) : S(i_1...,i_k) = T(i'_1...,i'_k) \Leftrightarrow$$

$$\Leftrightarrow i_k = i'_k \qquad (9)$$

In this expression, S is a read operation or a write operation and T is a write operation from a loop to be parallelized (S and T can be the same write operation). An example of a loop where this cutting technique is essential is the "tomcatv" benchmark of SPEC95 package.

3.4 Advanced techniques for the automatic parallelization

It is very common for results calculated in a loop to be used after the loop completes (that is, a DU-edge exits from the loop). For such loops, it is impossible to apply the basic technique because additional transformations have to be done. Figure 9 gives an example. In this figure, a variable *b* is changed under a condition. The loop is parallelized by the basic variable as before, but in this technique, the value of variable *b* is stored in the array *outdu* in each thread and a control flag (array *thread_act*) is set. The value of variable *b* is computed from the value that was calculated by the first thread or by the second thread (depending on the control flag). To avoid hidden inductions, variables with outgoing DU-edges must not be used in other locations within the loop.

Usually, loops have a few inductions besides the basic induction. To parallelize such loops, we developed an additional technique. This technique makes it possible to parallelize a few loop inductions, but these inductions must be reducible and must be presented like:

$$var = oper(var, step) \qquad (10)$$

In this expression, *oper* is a function: add, sub, multiplication, OR, AND, XOR, MAX (maximum) or MIN (minimum). Figure 10 gives an example of parallelization for such loops. In this technique, each induction occurs parallel on different threads. The loop containing inductions is parallelized by the basic induction. A result of each induction is stored in specially allocated arrays in the different threads. The threads calculate the final values of inductions based on these arrays built in different threads. This technique allows recognition of different types of inductions. Different transformations apply to different induction types. For example, if an induction is sum, then in the split procedure the corresponding induction variable will be initialized to 0, and the main thread will summarize the results of the induction calculated by the first thread and by the second thread.

4 Results

We used different benchmarks to estimate the effectiveness of automatic parallelization. Table 2 gives the best results

Table 2. The detailed information about execution of parallelized benchmarks.

BENCH	REAL	MEM_IMP	EXEC	L1 MISS	APB MISS	NOP	NO_COM
tomcatv	**1,37**	1,75	**1,48**	1,46	1,40	1,37	1,36
swim	**1,49**	1,5	1,96	1,94	**1,49**	1,49	1,48
mgrid	**1,61**	1,87	1,98	**1,85**	**1,65**	1,63	1,62
hydro2d	**1,30**	1,47	1,71	1,68	**1,26**	1,25	1,25
art	**1,37**	1,63	**1,46**	1,38	1,45	1,41	1,41

of speedups gained from automatic parallelization on the packages SPEC95 and SPEC2000. We ran the rests on Elbrus 3M1 under Linux 2.6.14. For the comparison, we did three different tests for each benchmark. The first test executed the serial version of a benchmark compiled with peak optimization options ("serial"). The second test executed the parallelized version of the benchmark compiled with peak optimization options and the "autopar" option ("autopar"). The third test executed the serial version of the benchmark on different processors at the same time ("parallel execution"). The last test makes it possible to estimate the maximum possible speedup gained from automatic parallelization, taking memory subsystem impact into account. It is obvious that the execution time of an automatically parallelized benchmark can theoretically be half that of the serial version. However, in parallelized benchmarks special conflicts occur in the memory subsystem despite the high throughput. These conflicts can be observed when different threads access to the same memory bank at the same time. Thus, it is difficult to achieve the factor-of-two speedup with automatic parallelization when both microprocessors are trying to fetch data from memory at the same time. The maximum possible speedup varies depending on the task. This speedup depends on memory access intensity in each task.

We determined the amount of executed explicit instructions (instruction words), different types of stalls, and empty instructions using specific profiling tools (Table 2). For each benchmark, Table 2 gives the ratios of number of executed explicit instructions (including different types of stalls and empty instructions) in the serial version to the same quantity for the automatically parallelized version on the first thread. In this table:

1. "REAL" is the achieved speedup of automatically parallelized benchmarks compared to serial versions;

2. "MEM_IMP" estimates the maximum possible speedup that could be gained from automatic parallelization, taking into account memory subsystem impact;

3. "EXEC" - ratios of number of executed explicit instructions (without different types of stalls and empty instructions) in the serial version to the same quantity in the automatically parallelized version on the first thread;

4. "L1 misses" - ratios account for "EXEC" with the number of pipeline stalls mostly because of L1 cache misses while the execution of load instructions;

5. "APB misses" ratios account for "EXEC", "L1 misses" with the number of pipeline stalls mostly because of L2 and APB cache misses;

6. "NOP" ratios account for "EXEC", "L1 misses", "APB misses" with the number of empty explicit instructions;

7. "NO COMMAND" ratios account for "EXEC", "L1 misses", "APB misses", "NOP" with the number of pipeline stalls because of Instruction Buffer cache misses.

These ratios are based on the results achieved in the first thread, as a parallelized version always waits on the second thread before the parallelized program completes.

An "EXEC" ratio (or "MEM_IMP" if it is less than "EXEC") for a benchmark corresponds to the speedup that can be achieved using the automatic parallelization functionality we described. Each ratio ("EXEC") for a benchmark represents a decrease in the number of explicit instructions executed in the first thread compared to the serial version. However, if in the automatically parallelized versions of benchmarks, many "L1 misses" or "APB misses" occur (or they are equal to the number of "misses" observed for serial versions), then the real speedup will be less. This is observed in all parallelized benchmarks. "L1 misses" and "APB misses" occur mostly when an accessed datum is not in the L1 cache or the APB buffer (misses). As result, these data have to be fetched from memory, causing pipeline stalls (blocking the microprocessor). From another point of view, in the automatically parallelized versions of the benchmarks the number of "L1 misses" or "APB misses" can decrease. Thus, the real speedup will be greater than the ratio of the number of executed explicit instructions in the serial version to the number of executed explicit instructions in the automatically parallelized version on the first thread. For example, in the parallelized version of the "art" benchmark, there were more "APB misses" than in the serial version.

An increased number of explicit empty instructions ("NOP") in an automatically parallelized benchmark could be connected with non-uniform distribution of instructions to long instruction words or excessive synchronizations.

5 Conclusion

In this paper, we presented our implementation of automatic parallelization in the Elbrus optimizing compiler. Automatic parallelization occurs during the earliest phases of the optimizing compiler, after loop optimizations, to simplify program analysis and transformation. Parallel loops are split into separate procedures to avoid changing the compiler's intermediate representation. Changing the compiler's intermediate representation could require redesigning all compiler optimizations, including the instruction scheduling phase. Such a restructuring could adversely affect the performance of the entire optimizing compiler. The performance of optimizations is especially crucial on an In-Order EPIC architecture.

We proposed several different parallelization techniques. The basic technique makes possible to parallelize loops in canonical form when there are no cross-iteration dependencies. To improve the effectiveness of parallelization and reduce synchronization costs, we proposed the specific technique of parallel loop cutting. We also considered an advanced techniques to parallelize loops in a different form.

We evaluated our techniques on benchmarks from SPEC95 and SPEC2000 packages. Our results showed the implemented automatic parallelization methods are generally effective. However, we did not achieve the maximum possible speedups because of "L1 misses" and "APB misses" in the parallelized benchmarks. These "misses" are concerned with excessive conflicts, which occur in the memory subsystem when different threads access to the same memory bank at the same time. It should be mentioned that the benchmarks are compiled to access to memory with a high intensity, which is closer to the maximum throughput. But even for the parallelized versions (where memory access intensity is doubled) the memory throughput is enough. The problem of observed conflicts is a subject of further investigation.

References

[1] B. Babayan, "E2k technology and implementation," in *Euro-Par '00: Proceedings from the 6th International Euro-Par Conference on Parallel Processing*. London, UK: Springer-Verlag, 2000, pp. 18–21.

[2] M. S. Schlansker, B. R. Rau, and Multitemplate, "Epic: An architecture for instruction-level parallel processors," HP labs, Tech. Rep., 2000.

[3] V. Volkonskiy and A. Kim, "Parallelizm ideas development in the elbrus series computer architecture," in *International Conference on Parallel Computations and Control Problems*. Moscow,Russia: Institute of Control Sciences, Rissian Academy of Sciences, 2008.

[4] S. A. McKee, "Reflections on the memory wall," in *CF '04: Proceedings of the 1st conference on Computing frontiers*. New York, NY, USA: ACM, 2004, p. 162.

[5] S. S. Muchnick, *Advanced compiler design and implementation*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1997.

[6] A. Aiken, A. Nicolau, and S. Novack, "Resource-constrained software pipelining," *IEEE Trans. Parallel Distrib. Syst.*, vol. 6, no. 12, pp. 1248–1270, 1995.

[7] N. Snavely, S. Debray, and G. Andrews, "Predicate analysis and if-conversion in an itanium link-time optimizer," in *Proceedings of the Workshop on Explicitly Parallel Instruction Set (EPIC) Architectures and Compilation Techniques (EPIC-2)*, 2002.

[8] X. Tian and M. Girkar, "Effect of optimizations on performance of openmp programs," in *Proceedings of the 11th International Conference on High Performance Computing*. Berlin,Germany: Springer Berlin / Heidelberg, 2004, pp. 133–143.

[9] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers: principles, techniques, and tools*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1986.

[10] K. Kennedy and J. R. Allen, *Optimizing compilers for modern architectures: a dependence-based approach*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2002.

[11] D. E. Maydan, J. L. Hennessy, and M. S. Lam, "Efficient and exact data dependence analysis," in *PLDI '91: Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation*. New York, NY, USA: ACM, 1991, pp. 1–14.

[12] P. Tirumalai, M. Lee, and M. Schlansker, "Parallelization of loops with exits on pipelined architectures," in *Supercomputing '90: Proceedings of the 1990 ACM/IEEE conference on Supercomputing*. Los Alamitos, CA, USA: IEEE Computer Society Press, 1990, pp. 200–212.

[13] W. Pugh, "A practical algorithm for exact array dependence analysis," *Commun. ACM*, vol. 35, no. 8, pp. 102–114, 1992.

[14] M. Wolfe and C. W. Tseng, "The power test for data dependence," *IEEE Trans. Parallel Distrib. Syst.*, vol. 3, no. 5, pp. 591–601, 1992.