Increasing Effective IPC by Exploiting Distant Parallelism

Iván Martel, Daniel Ortega, Eduard Ayguadé and Mateo Valero
Departament d'Arquitectura de Computadors,
Universitat Politècnica de Catalunya – Barcelona, Spain
e-mail: {imartel,dortega,eduard,mateo}@ac.upc.es

Abstract

The main objective of compiler and processor designers is to effectively exploit the instruction–level parallelism (ILP) available in applications. Although most of the times their research activities have been conducted separately, we believe that a stronger co–operation between them will make effective the increase of potential ILP coming from future architectures. Nowadays, most computer architecture achievements proceed towards the overcoming of the hurdle imposed by dependencies in the code, by means of extracting parallelism from large instruction windows. However, implementation constraints limit the size of this window and therefore the visibility of the program structure at run–time.

In this paper we show the existence of distant parallelism that future compilers could detect. By distant parallelism we mean parallelism that can not be captured by the processor instruction window and that can produce threads suitable for parallel execution in a multithreaded processor. Although this parallelism also exists in numerical applications (going far beyond classical loop parallelism and usually known as task parallelism), we focus on non–numerical applications, where the data and computation structures make difficult the detection of concurrent threads of execution.

Some preliminary but encouraging results are presented in the paper, reporting speed–ups in the range of 1.2 to 2.65. These results seem promising and want to show a new insight in the detection of threads for current and future multithreaded architectures. It is important to notice at this point that the benefits described herein are totally orthogonal to any other architectural techniques targeting a single thread.

1 Introduction

The parallelism exhibited by programs depends not only on the program execution model but also on the architecture under which they are executed. Many theoretical or limit studies have focused on analysing the available parallelism in programs under different architectural constraints. One of the first studies [20] reported average IPCs (Instructions per Cycle) between 2 and 3. However, this study did not speculate control in any way, reducing the window from which to extract parallelism to just a single basic block.

Later studies [15] showed the importance of branch prediction in order to increase IPC. This technique allows the exploitation of large amounts of ILP (Instruction–level parallelism) by looking for parallelism across basic block boundaries. Doing so implies the speculation of control which relies on the effectiveness of branch prediction schemes.

More recent limit studies have focused on removing false dependencies (either between registers or memory locations) [7, 26], analysing their effects and discussing the feasibility of removing them. The importance of register renaming is crucial in order to take advantage of the parallelism in programs [21].

Recent papers [14] show that not only false dependencies due to the reuse of storage locations limit the parallelism that can be exploited. In their research, other compiler induced dependencies are also investigated. This kind of dependencies, dynamically detected as true data dependencies by the architecture, are not introduced by the algorithm itself, but by the way the compiler expresses computation; some of them can be avoided by using different code generation techniques.

Sometimes these limit studies have induced architectural proposals that try to accomplish the theoretical IPCs observed. However, this is not always possible because of the characteristics of the study itself, that totally relax certain architectural constraints, (e.g. perfect branch prediction or unbounded resources). Even without assuming limit conditions, the proposals may not be worth implementing, such as very large instruction windows [11].

The main way of increasing IPC, and therefore speeding up applications, has always been the exploitation of the inherent parallelism in programs, either using software techniques or hardware mechanisms. Although the majority of previous research in ILP focused on the performance of a single thread of execution, a more effective increase of ILP can be achieved from the execution of multiple threads from the same program. We strongly believe that this increment in IPC should arrive from a combined effort from the design of algorithms, compiler techniques and computer architecture. In any case, the extraction of parallelism from programs is not an easy task and is based on the analysis and detection of data and control dependencies.

There have been several proposals intended at the overcoming of the data and control dependencies. As said before, register renaming may efficiently overcome false data dependencies across registers. Recent proposals try to predict values to break down true data dependence chains and therefore expose more parallelism [8]. To be able of exploiting higher degrees of ILP it is necessary to look for parallelism across basic block boundaries and support from effective branch prediction schemes [17, 28]. This control speculation allows the simultaneous execution of instructions from different basic blocks and has originated some novel architectures like the ones in the multiscalar [18] and trace [16] processors.

In order to further increase the number of instructions from which to exploit parallelism, multithreaded architectures [27, 23] have been proposed. Threads coming from the same application are usually found in parallel loops detected by the compiler [22]. Hardware mechanisms have been proposed to detect dependence violations when loops, whose data dependence patterns cannot be decided at compile time, are executed speculatively as parallel loops [19]. Other proposals try to dynamically detect these loops and extract their semantic information at run–time [9]. Notice that this level of detection of loops seems to be the frontier between hardware and software mechanisms. Parallel loops are much better recognised by software while other loop structures, more complex or with unpredictable dependence patterns, have to be detected or speculated via hardware. The work presented in [24] goes even a bit further by speculating data dependencies between a loop and its continuation.

Programs usually have much more parallelism than what the hardware can dynamically detect. The hardware is restricted to 'see' a tiny portion of the program being executed, because of the limitations of its instruction window and the limited semantics of the instructions. In our work, we intend to exploit non–structured thread parallelism that could be statically extracted from the source code by the compiler. In addition to the loop–level parallelism detected by current parallelising compilers (like POLARIS [3] or SUIF [6]), non–structured parallelism can also be detected when accurately combining the analysis of control and data dependencies in a hierarchical task graph [12] (like in the Parafrase–2 [13] or PROMIS compilers [4]). However, applications sometimes show parallelism between zones of code very distant apart (in terms of number of instructions executed between them) that can not be automatically detected by the compiler because of the limited scope of its analysis techniques or because it is hidden by the data and computation structures used in the application. Many numerical applications also show multiple levels of parallelism, combining both task parallelism (usually at the coarser level) and loop–level parallelism [1, 2].

Non–numerical applications also have non–homogeneous parallelism between zones distant apart. Nonetheless, the achievement of this parallelism may imply the use of parallelising techniques, analogous to the ones already used for parallelising loops, but in a more complex way. Therefore, the first objective of this paper is to demonstrate that non–numerical applications show high degrees of thread–level parallelism, and that remarkable benefits can be obtained by exploiting it. The second objective of this paper is to show some of the compiler transformations (similar to the ones currently applied for the parallelisation of numerical applications) that would be required to exploit this parallelism. Four benchmarks from SPEC95int are used (`compress`, `m88ksim`, `go` and `ijpeg`). We manually generate threads for them (using standard thread creation and synchronisation system calls). We use an execution–driven environment to simulate the execution of these threads on an ideal processor. The preliminary performance figures reported for this ideal processor try to reveal sources of parallelism in non–numerical applications that will probably never be discovered at run–time and worth to be detected by future parallelising compilers.

The organisation of the paper is as follows. Section 2 presents a description of the types of parallelism that can be found in both numerical and non–numerical applications. Section 3 describes in detail the analysis of each of the four benchmarks from SPEC95int. Section 4 summarizes the compiler transformations used for their parallelisation. Section 5 describes the simulation environment and presents results obtained from this simulation on an ideal architecture. The paper ends with the conclusions in Section 6.

2 Parallelism in programs

The existing parallelism in programs can be classified according to the quantity of instructions it covers: instruction level parallelism (ILP) and thread level parallelism (TLP). ILP is accomplished via the concurrent execution of instructions belonging to the same flow of control. TLP differs from ILP in that what is considered totally parallel are groups of instructions, despite the fact that instructions belonging to a particular group may have dependencies among them. TLP offers advantages in the sense that different types of threads can co–exist at a time in the processor, balancing the needs for different resources; however, inter–threads conflicts in the memory hierarchy can also reduce the final performance. Nevertheless, the benefits derived from TLP are orthogonal to those coming from ILP, what makes the combination of both techniques advisable and desirable.

On the contrary to ILP, TLP is hardly obtained via hardware. Some researchers have proposed to speculate zones of execution with a high probability of being parallel, thus achieving TLP; nevertheless, these zones must be locally adjacent and completely parallel at an instruction level, still leaving higher levels of parallelism to be found at compile time. The amount of TLP found at compile time exclusively by compilers is very small, and is mainly found in numerical applications at the level of loops. Non–numerical applications are, in general, considered non parallelisable because of the data and computation structures used in this kind of applications. Normally, compilers need the help of programmers by means of directives and assertions, multithreading libraries, or restructuration of the source code to make parallelism available to them.

In the following subsections we analyse different forms of TLP that are not usually detected by current parallelising compilers, either for numerical and non–numerical programs. Threads detected encapsulate regions of code that can be executed in parallel and that are distant in terms of number of instructions (statically in the original source code and/or dynamically when they are executed). Later in the next section we focus on the parallelisation of some non–numerical applications from SPEC. Additional results for some numerical SPEC applications can be found in the extended version of this paper [10].

2.1 Numerical applications

Existing compiler techniques for finding parallelism in numerical applications refer primarily to loops. Different techniques have been proposed to analyse and transform codes in order to make loops totally parallel. Although other levels of parallelism can exist in numerical applications (usually

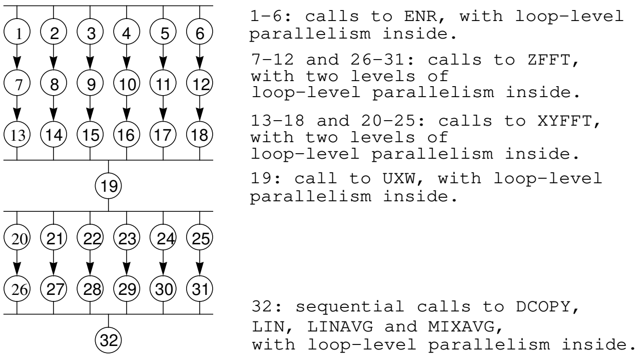

Figure 1: Task graph for the SPEC95fp `Turb3D` program.

at the level of parallel tasks), compilers usually fail in automatically detecting them. The need of accurate interprocedural analysis and optimisation techniques applied across procedure boundaries and data equivalencies are needed to successfully find these sources of parallelism.

The exploitation of multiple levels of parallelism in numerical application is also an issue to consider. Loop–level parallelism sometimes produces poor scalability because the amount of computation is too small or because although the theoretical computation is high, the data movement overheads tend to hide the benefits of the parallel execution. Exploiting multiple levels of parallelism may distribute other sources of parallelism among groups of processors and therefore avoid the negative effects of scalability. In addition to that, the overheads related to thread creation and joining could be reduced considerably if higher (coarser) levels of parallelism are exploited.

Although certain combinations of application and architecture do not need new sources of parallelism, other combinations can benefit from the exploitation of multiple levels of parallelism (for example, clustered architectures in which the processors in a cluster can exploit loop–level parallelism and outer levels of parallelism can be exploited among clusters).

Notice that, in any case, parallelism found in numerical applications is usually well structured (in the sense that threads have assigned the same kind of computation) and synchronisation between them occurs at global points (related to all threads or to a group of them) by means of barriers. Most current proposals for parallel constructs (like the ones in the OpenMP [5] extensions to Fortran and C/C++) are designed having in mind this kind of parallelism.

For example, `Turb3D` is a program from SPEC95fp benchmark suite that simulates isotropic homogeneous turbulence in a three dimensional cube. The application works primarily with six different tridimensional matrices, called u, v, w, ox, oy and oz. Its main function, *turb3d*, contains an iterative time–step loop that consists of 4 loops, a call to uxw plus some additional calls to implement the timestepping scheme. These four loops are parallel and make the same operation over the six matrices, being different in the way each loop accesses them. The functions called from each of the loops, *zfft* and *xyfft* perform the FFT transformation over the different matrices, from time domain to frequency domain and viceversa. Between the first two loops and the last two ones function uxw combines contributions from all matrices in order to produce new versions for some of them. Although not having parallelism at the level of different matrices, uxw has parallelism at the level of loops. Similarly, the routines invoked to implement the time stepping scheme also have parallelism at the level of loops. Figure 1 summarises the parallelism structure of this application.

In the extended version of this paper [10] we show the possible speed–ups for this application when combining different levels of parallelism. The highest speed–up achieved is 391.05 when all levels of parallelism are opened. Nevertheless, most of this speed–up (309.78) is achieved when two levels are opened, the highest one at the level of sections and the outer loop level. This behaviour has also been noted in the other application reported in the extended version of this paper.

2.2 Non–numerical applications

Numerical applications usually have regular control and data structures that can be easily understood by the compiler. Nevertheless, non–numerical applications use dynamically allocated data structures (such as pointers, lists and trees) accessed through one or several levels of indirections, thus complicating the task of the compiler. Therefore, small amounts of parallelism are expected from non–numerical applications, and in general they are considered to be single threaded.

The main constraint we imposed ourselves when looking for distant parallelism in the non–numerical applications was to avoid any change of the algorithm originally implemented in SPEC95 that made them more amenable for parallelisation. It is also important to remark at this point that the purpose is to show the existence of distant parallelism in these applications; this has usually implied to skip any possible parallelism that could be easily captured by current processor execution windows.

The exploitation of distant parallelism has been based on finding zones of code whose contents are semantically parallel. As we will see in Section 3, parallelism found in non–numerical applications tends to be non–homogeneous and non–structured. Possible false data dependencies among zones will be eliminated by applying techniques similar to the ones currently used in the parallelisation of numerical codes (for instance, privatisation of simple and structured variables). Possible true data dependencies sometimes will require a producer–consumer(s) co–ordination scheme, reduction operations around complex data structures and very frequently the need of synchronisation between the threads at non regular points of execution.

For example, Figure 2 shows one of the strategies that has been used to parallelise some of the benchmarks. The parallelisation strategy is based on a thread that produces

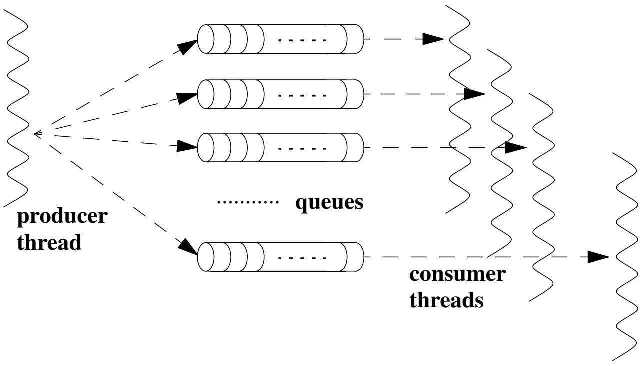

Figure 2: Parallelisation strategy based on the One Producer–Many Consumers paradigm.

data to be consumed by a set of consumer threads; in this case, a set of queues are used to buffer the data being transmitted from producer to consumers threads. This strategy has been used for the parallelisation of `compress` and `ijpeg`. Other strategies based on the simultaneous execution of (data and control) dependent or independent threads have been used in `m88ksim`, `go` and `ijpeg`.

3 Particular parallelisations

All the programs whose parallelisation is described in this section belong to the SPEC95int benchmark suite. Although we just describe four benchmarks, we believe that the techniques described are representative and applicable to the rest of them. For each benchmark we describe the application itself, the parallelisation structure and its potential benefits.

3.1 M88ksim

Description of the program
The `m88ksim` program is a simulator of the Motorola 88100 processor. The program implements the simulation of the processor datapath and a user–oriented environment for debugging, including the use of core files. It spends most of the execution time in the simulation phase.

The function that begins the execution of the simulated program is named *go* and iteratively calls the *Data_path* function for every simulated instruction. With a complex structure, *Data_path* simulates the data cache, the instruction cache, the memory management units, the functional units, the timing and verifies the triggering of breakpoints. But all those tasks are performed in an interlaced fashion. First, the function *ckbrkpts* checks if the current program counter stands for a code breakpoint (if it does, function *Data_path* returns). Then *cmmu_function1* simulates the behaviour of the instruction cache memory management unit, in spite of the actual instruction that is obtained later by calling *getmemptr*.

Once the simulator has a new instruction, it analyses the availability of the operands and resources, and then, a large case statement evaluates the operation code, performing the corresponding action. If it is a memory reference the data cache is simulated, and a breakpoint checking is performed. Finally *killtime* updates the time in every structure according to the instruction latency and the program counter is modified.

Structure of the parallelisation
Basically, there are three code sections that can be executed simultaneously with minimum data conflicts among them. These sections belong mainly to the *Data_path* function and are generated after applying code motion within and across procedure boundaries. These sections are shown in Figure 3 under the names of *timing*, *exe* and *fetch*, representing the timing, the execution and the fetch of the next instruction, respectively.

Three threads are used to simulate the timing, two for the execution and one for the fetch mechanism. The threads of the timing are executing primarily the section of the code belonging to the function *killtime*. They assume that the variable *cmmutime* equals zero, which occurs nearly always except when a memory instruction misses in the data cache. This will increase the timing, and is represented in the figure by the dotted arrow called *cmmutime*. Breakpoints are checked during the execution phase. As breakpoints rarely occur, this thread is not always executed.

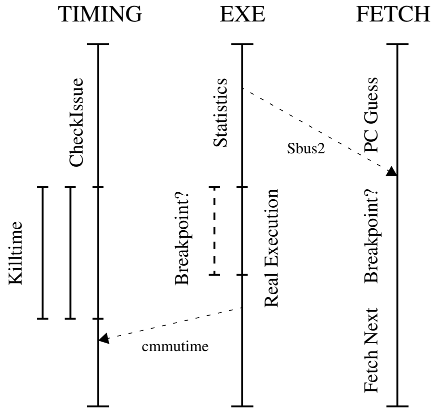

Figure 3: Structure of the threaded parallelisation in `m88ksim`.

Potential benefits
The execution of function *Data_path* represents 90% of the total execution time when this program is executed with the test input. Every simulated Motorola instruction takes around 1,200 machine instructions of the host processor. The critical path that limits the performance is the timing simulation. It takes approximately 360 instructions. Therefore, the theoretical speed–up that can be achieved is 2.70.

3.2 Compress

Description of the program
The program that comes with the SPEC95 benchmark suite is a modification of the original UNIX compression algorithm. Its main loop consists on 25 iterations of successive compressions and decompressions of nearly the same data. The data is slightly changed each iteration by adding characters at the end of it, but the amount of characters added is very small in comparison with the total amount, representing less than 2 per ten thousands.

During the compression it is possible to detect repetitions of the same pattern of computation. The compression algorithm is an implementation of the LZW algorithm which uses a translation table made up on the fly both in the compression and in the decompression phase. The compression algorithm is made up of mainly two different functions, *compress* and *output*. *Compress* takes one character from the input and merges it with the previous one; if this conjunction is found in the table, it takes its code and scans another character from the input. Sooner or later the conjunction of characters will not appear in the table, meaning that it has not appeared yet in the input data, and the *compress* function will introduce it in the table and produce a unique output code for it. The significant bits of these output codes range from 9 to 16 bits, and to benefit more from the compression, they are packed by the function *output*. Eventually, the output codes will run out, and at this moment, the program enters a repetitive task of looking each 10,000 outputs the compression rate. When this rate decreases, the current table is considered useless, and a special code meaning table cleaning is produced. The process starts again with a new table.

The decompression phase has mainly two different functions, *decompress* and *getcode*. *Getcode* is a function that takes the input of the decompression phase (that is, the compressed data) and unpacks the codes, giving them unpacked to the decompress phase, which does the inverse method done by *compress*, it searches with these codes in the table, producing the primary input.

Structure of the parallelisation
We have parallelised both the compression phase and the decompression phase. In each of them two threads have been created, in a producer consumer way (Figure 2). In the compression phase function *compress* acts as the producer of codes and the *output* function acts as the consumer of codes. The producer thread passes the codes to the consumer thread through an intermediate queue, thus allowing the producer to work ahead of the consumer. The parallelisation also required to apply some code motion and privatisation of variables.

In the decompression phase the function *getcode* has been made the producer of codes and the function *decompress* the consumer. We will see later that this makes the parallelisation work better, because the smallest thread is the producer and the larger one is the consumer.

Potential benefits
In the sequential compression there are 81,396 calls to the function *output* before the first cleaning of the table occurs. Knowing that this sequential compression is 24,703,595 instructions long, it can be said, grosso modo, that the cycle of compression is 303.49 instructions in average. We have decided to call a cycle of compression the amount of work done by the function *compress* in order to get a code, pass it to output and the work done by *output* to pack it.

After generating the threads, the number of instructions executed in function *compress* is 19,995,496 and in function *output* is 5,059,462; therefore, the average instructions/cycle of compression is 245.65 for function *compress* and 62.15 for function *output*. If both threads could execute in parallel without any problem, the critical path would be reduced from 303.49 to 245.65, thus achieving a theoretical speed–up of 1.24.

An equivalent analysis for the decompression phase has also been done. The cycle of decompression is analogous to the cycle of compression, and according to the amount of instructions executed (13,778,302) and the amount of cycles (81,396, the same as in the compression phase) an average time in instructions per cycle of decompression for this phase can be extracted, which is 169.27 instruction/cycle.

After generating the threads, the number of instructions executed in function *decompress* is 8,018,357 and in function *getcode* is 5,366,955. This makes 98.5 instructions per cycle of decompression for *decompress* and 65.93 instructions per cycle of decompression for *getcode*. If both threads could be executed in parallel without any problem, the critical path would have been reduced from 169.27 to 98.51 instructions per cycle, thus achieving a theoretical speed–up of 1.72.

3.3 Go

Description of the program
The program that comes with the SPEC95 benchmark suite is a modified version of the go playing game. The modified version of the program allows the selection of the skill level, the size of the board, and the introduction of a set of moves to start with.

At the highest level, the program spends the biggest portion of time executing function *life*, which analyses characteristics of a particular distribution of stones in the board. *Life* calls different functions that gather heuristic information which will help decide the best move in every turn. Many of these functions call *iscaptured* which does a tactical analysis of a group, spending around 80-90% of the total execution time. A group is a set of stones which potentially controls a portion of the board.

Iscaptured modifies a big amount of the data stored in lists. The underlying code implements a general artificial intelligence algorithm that evaluates a speculative tree of moves. As the function gets deeper into the tree it must update the structures locally in order to reflect the new game state. In the same way, it must recover the state when returning from any node to the father. Finally, the algorithm goes back to its initial state, returning a condition that is used to rearrange group armies or to detect critical spots.

Structure of the parallelisation
At a low level, the program spends most of the time in functions that manage lists. Nevertheless, we have focused our parallelisation at a coarser level, in functions that gather heuristic information. In particular we have parallelised functions *bdead* and *findcaptured* which mainly contain a loop that calls *iscaptured*.

Several instances of *iscaptured* could be executed in parallel if local structures were privatised and if it could be guaranteed the exclusive accesses to some lists. The loops that contain the call to *iscaptured* also contains instructions that prevent the parallelisation. Loop distribution is applied in order to separate the parallel zone from the sequential one. The parallel part is the one containing a variable amount of calls to *iscaptured*, which are executed in separate threads with private structures.

To overcome the problem of the accesses to lists, all modifications to lists are done locally in *iscaptured*. In the sequential part of the loop, these local modifications to lists are made global in a reduction scheme.

Potential benefits
The loops parallelised are the main ones in *findcaptured* and *bdead*. They take, respectively, 13% and 37% of the program time, mainly in *iscaptured*. As an upper bound we could expect a local speedup equal to the mean number of calls to *iscaptured* in each function, 9.7 and 5.7 respectively, although this number varies among different games. This ideal case should bring a global speedup of 1.7. There is little penalty in the parallelisation because the threads are large enough. Furthermore, the second phase of the loops (the sequential part) is very little in comparison with the time consumed by *iscaptured*. However, the local speedups achieved are smaller than the ideal case. The reason is the load unbalance due to the variation in execution time of *iscaptured* that will be discussed later.

3.4 Ijpeg

Description of the program
The program that comes with the SPEC95 benchmark suite is a version of IJG JPEG application that compresses and decompresses at multiple settings a previously loaded-into-memory bitmap image and produces statistics about the whole process. Conceptually it could be seen as a search for the optimal compression parameters program, although no attempt is made to determine any quality/size trade–off.

The image is represented by three colour matrices defining RGB colours. The image is converted from this space colour to a luminance–chrominance space colour and then transformed into a frequency space via discrete cosine transforms. This new image is compressed with a Huffman encoding. This process conforms the jpeg compression. An inverse transformation conforms the decompression.

Structure of the parallelisation
Four different parts of the program have been analysed and parallelised. They cover over 60% of a standard execution of the benchmark. Two of these parts have been parallelised using a fixed number of threads while the other two work in a producer–consumers way, thus having a parametrisable number of threads.

One of the zones parallelised is the conversion from RGB to YCC in the function *rgb_ycc_convert*. Three threads have been extracted from this code, each of them working with a particular colour. Similarly, function *h2v2_merged_upsample* also uses the same type of parallelisation, although the conversion of colours is done the other way round. In both parallelisations only two threads are created, leaving the rest of the work to be done by the main thread.

A different parallelisation strategy has been used in function *forward_DCT*. This function iterates through all the blocks in the image performing the DCT and afterwards descales the coefficients and stores them in the appropriate structures. We have used the producer–consumers paradigm (Figure 2) in this parallelisation, where the main thread is in charge of distributing the blocks among the consumer threads. Similarly, function *jpeg_idct_islow* also consists of a main thread and a parametrisable number of consumer threads that are in charge of making the DCT computation.

Potential benefits
We have theoretically analysed the potential benefits of the parallelisation in this program according to profile information and the knowledge we have of the different zones. If we consider each of the zones parallelised to have decreased by a factor equal to the number of threads, then we can estimate the total speed–up obtainable from the profile information.

The potential benefits of the parallelisation of the first two zones explained, those dealing with colour transformation, can be found by dividing its critical path by three. The other two zones described have more parallel threads, thus potentially achieving bigger reduction of their critical path.

We have analysed profile information from two different runs of this benchmark. The first one uses the standard input file (penguin.ppm), while the second one uses the test input file (specmun.ppm). The profile information differs from one to the other sensitively. The bigger input offers much more potential than the small one. The amount of code parallelised covers 63% and 57%, respectively with the change of input. The time spent in each of the zones parallelised also decreases with the input file, therefore the potential benefits change when speaking of the bigger input or the smaller one.

With the bigger input we have calculated a potential speed–up of up to 2.04 with 16 threads, while the smaller input only shows a potential speed–up of 1.70 with the same number of threads. Nevertheless, both analysis show that most of the improvement is achieved when using up to 8 threads. The speed–up values for 4, 8, 12 and 16 threads are 1.796, 1.953, 2.012 and 2.043, respectively.

4 Compiler techniques

In this section we summarize the compiler transformations applied in the parallelisation of the programs described in Section 3. All of them assume an accurate interprocedural analysis able to disambiguate memory references and efficiently derive alias information. In addition to that, the compiler framework should also be able to apply some transformations which have been thoroughly studied in the field of parallelising compilers for numerical codes.

Most of the programs required some kind of code movement both within the scope of a procedure and across procedure boundaries. Sometimes this has been applied to simply balance the amount of work done within threads; this would require the estimation of execution costs, either through program profiling or static estimation. In other situations code movement has been applied to isolate sequential from parallel parts. For instance, loop distribution has been applied to partially parallelise some of the loops that appear in `go`.

Variable privatisation has been extensively used in all the programs. This privatization involved scalar as well as structured variables (e.g. lists). The detection of reduction operations has also been applied in some of the programs to break recurrences. This implied the generation of private copies for the variables involved in the reduction and the sequential update to make their effect global. These reductions are usually applied to lists and other more complex data structures, which adds more difficulty to the process.

The semantic parallelism manually detected would require the construction of a task graph combining control and data dependences in the form of task precedences. A hierarchical definition for this graph and the above mentioned code motion together with cost estimation would enable the detection of efficient distant parallelism. Satisfying the dependences among the threads executing these tasks has been accomplished by means of point to point synchronisation (like in `m88ksim`), guaranteeing exclusive access to some data structures (like in `go`) or using the producer–consumers paradigm (like in `compress` or `ijpeg`). For the later, this would require realising that the task graph has 'narrow' zones (in terms of precedences) which represent the place in which the producer thread would pass information to the consumer thread. The compiler should introduce here the structures to communicate both threads and allow them to run asynchronously.

5 Experimental results

5.1 Simulation environment

In this section we describe the environment used for the simulation of the parallel execution of the non–numerical applications parallelised in Section 3. These parallelisations have been done using standard UNIX thread creation and synchronisation system calls. We have used the MINT execution–driven simulator [25] running on top of an SGI Origin2000 system.

Our simulation environment assumes that all instructions execute in one cycle and perfect memory (i.e. all loads and stores hit in cache). We do not take into account possible interferences that may adversely affect cache performance when running multiple threads.

MINT has the possibility of defining the costs in terms of instructions of all system calls, such as the ones used for thread creation, synchronisation and communication. In this paper we assume that they are executed in one cycle, because we believe that multithreaded architectures that will exploit this kind of parallelism will have instructions and architectural support to execute them.

Our current work targets the simulation of these parallel codes on a detailed processor simulator where all these parameters are taken into account.

5.2 Analysis of results

The results presented in this section will always refer to speed–ups, considering the speed–up of a program the total number of cycles the sequential version takes to complete divided by the total number of cycles of the parallelised version. The results always refer to complete executions.

5.2.1 M88ksim

The speed–up reported for the parallelisation of `m88ksim` is of 2.65. The simulation was done using the test input, which consists of 500K Motorola instructions, mainly logic and memory instructions. Similar results in the execution profiles are obtained with other input files corresponding to integer Motorola applications.

The difference with the theoretical speed–up (reported in Section 3) is due to the variance of the execution phase. Although its mean size is smaller than the mean size of the timing phase, it becomes dominant for certain kind of instructions. For example, when floating point instructions are simulated (10% of time in the test input set used) the execution thread grows up to 15,000 instructions, making impossible to find more tasks to perform in parallel. Memory instructions also tend to make the execution phase longer than the timing, reducing the potential speed–up. Sometimes, certain conditions prevent the execution of all the threads in parallel (i.e. when a trap occurs).

5.2.2 Compress

The speed–ups reported for the parallelisation of the compression and the decompression parts of the `compress` program are 1.22 and 1.52, respectively.

The difference relative to the theoretical values reported in Section 3 are because of the large variance in the length of the threads, especially in the producer thread in the compression. This variance correlates with the particular moment in the construction of the table of translation of codes. At the start of the compression phase, few codes have been introduced, therefore the threads are small. At later times, when more codes have been introduced in the table, threads become larger. The overhead introduced by the parallelisation itself (queue management) also reduces the potential speed–up achievable.

5.2.3 Go

As a result of the parallelisation described in Section 3, the speed–up obtained for functions *findcaptured* and *bdead* is 2.2 and 2.4, respectively. The parallelisation of these two functions results in a global speedup of 1.4 for the whole `go` application.

Notice that the speed–up obtained is smaller than the one predicted in Section 3. The reason for these unexpected results is the large variance in terms of execution time of function *iscaptured*. For example, Figure 4 shows the probability of executing a particular number of instructions in this function when called from routine *bdead* (which performs 150,377 calls to *iscaptured* when executes with play level 40 and a board size of 19). The shape of the plot is similar for other invokations from other routines. As a consequence, some threads take longer than other threads, reducing the potential benefits that a balanced execution could return.

5.2.4 Ijpeg

The speed–up reported for the parallelisation of program `ijpeg` depends on the number of threads (4, 8 and 12 threads) devoted to the execution of the consumer parts in two of the

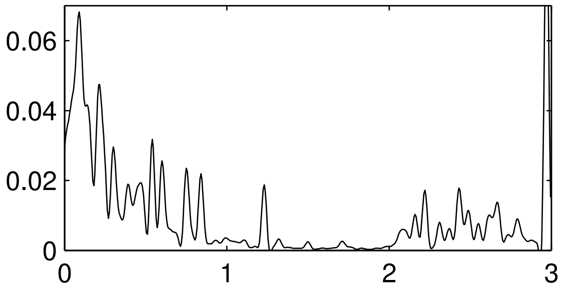

Figure 4: Probability of executing a particular number of instructions (x10^5, horizontal axis) when calling procedure *iscaptured*.

four functions parallelised. The other two parts are always executed with the same number of threads (three). The different versions evaluated are labeled according to the number of consumer threads.

We have run simulations with both the test input file, *specmun.ppm*, and the standard input file, *penguin.ppm*. Figure 5 shows the speed–up achieved for these two input files and for the three configurations mentioned above. Notice that the speed–up ranges from 1.37 to 1.44 with the test input and from 1.48 to 1.57 with the standard one.

Notice that the speed–ups reported are smaller than expected. This is due to an excessive redundancy in the computations done in some threads. This replication implies some threads read the same values and do the same computation to produce the same intermediate result. This redundancy has been introduced to avoid the overhead that would introduce the direct synchronisation of dependent threads.

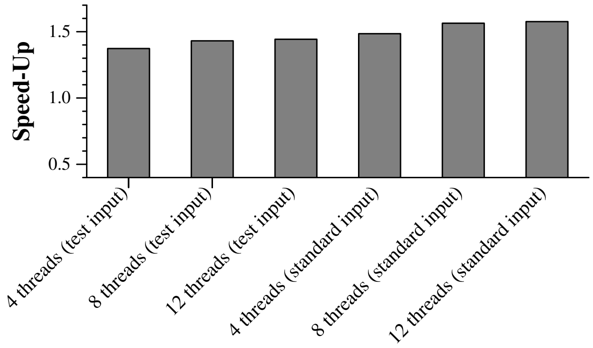

Figure 5: Speed–up for `ijpeg`.

6 Conclusions

The main way of increasing IPC, and therefore speeding up applications, has always been the exploitation of the inherent parallelism of programs, either using software techniques or hardware mechanisms. The majority of previous research in ILP focused on the performance of a single thread of execution; however, a more effective increase of ILP can be achieved from the execution of multiple threads belonging to the same application. Although several previous proposals have focused on the dynamic detection of these threads (around loops), we push for a combined effort, both from compiler and architecture, towards getting higher effective increments in IPC. The compiler should be able to detect distant parallelism (not captured by the hardware mechanisms included in the processor) and the processor should be able to efficiently exploit intra–thread parallelism and manage the multiple threads efficiently.

Parallel loops, which are currently at the frontier between hardware and software detection mechanisms, are the main sources of threads in numerical applications. The limited visibility of hardware mechanisms do not allow the exploitation of more distant parallelism that exists both in numerical and non–numerical applications. In this paper we have demonstrated that non–numerical applications can benefit from thread–level parallelism. We have used four SPEC95int applications (`compress`, `m88ksim`, `go` and `ijpeg`) to present different parallelisation strategies that requires minimum changes in the application. We have also shown that the compiler transformations applied are similar to the ones available in current parallelising compilers for numerical applications. The additional difficulty comes from the use of dynamically allocated data structures accessed through one or several levels of indirections; efficient and accurate interprocedural analysis techniques are required to overcome it.

The speed–ups reported by our simulations on a ideal processor (perfect memory and one cycle execution for all instructions) show promising increases in the range of 1.20 to 2.65. The results obtained are not to be seen as unreachable limits but as a new approach into the extraction of parallelism. The benefits described are orthogonal to any other architectural techniques focused on a single thread.

We consider this paper to be a first stage in a long term research in combining software and hardware techniques for effectively increasing IPC. We expect architectural proposals to derive from our current investigation.

7 Acknowlegments

This work was supported by the Ministry of Education of Spain under contracts CICYT TIC98–0511 and TIC97–1445–CE and grant AP98–42879678, the Direcció General de Recerca under grant 1998FI–00292–APTIND, and the CEPBA. The authors wish to thank Jesús Labarta, Jesús Corbal and Xavier Martorell for the time devoted to fruitful discussions and their help in understanding some of the benchmarks.

References

[1] E. Ayguadé, X. Martorell, J. Labarta, M. González, and N. Navarro. Exploiting parallelism through directives on the nano-threads programming model. *10th International Workshop on Languages and Compilers for Parallel Computing*, August 1997.

[2] H.E. Bal and M. Haines. Approaches for integrating task and data parallelism. *IEEE Concurrency*, July-September 1998.

[3] W. Blume, R. Eigenmann, J. Hoeflinger, D. Padua, P. Petersen, L. Rauchwerger, and P. Tu. Automatic detection of parallelism: A grand challenge for high performance computing. *IEEE Parallel and Distributed Technology*, Fall 1994.

[4] C.J. Brownhill, A. Nicolau, S. Novack, and C.D. Polychronopoulos. The promis compiler prototype. *1997 Conference on Parallel Architectures and Compilation Techniques*, June 1997.

[5] v. 1.0. Fortran Language Specification. Openmp organization. *www.openmp.org/openmp/mp-documents/fspec.ps*, October 1997.

[6] M.W. Hall, J.M. Anderson, S.P. Amarasinghe, B.R. Murphy, S.W. Liao, E. Bugnion, and M.S. Lam. Maximizing multiprocessor performance with the suif compiler. *IEEE Computer*, December 1996.

[7] N.P. Jouppi and D.W. Wall. Available instruction-level parallelism for superscalar and superpipelined machines. *3th International Conference on Architectural Support for Programming Languages and Operating Systems*, May 1989.

[8] M.H. Lipasti and J.P. Shen. Exceeding the dataflow limit via value prediction. *29th Annual International Symposium on Microarchitecture*, December 1996.

[9] P. Marcuello and A. González. Speculative multithreaded processors. *ACM International Conference on Supercomputing*, 1998.

[10] I. Martel, D. Ortega, E. Ayguadé, and M. Valero. Increasing effective ipc by exploiting distant parallelism. Technical Report UPC-DAC-1998-59, Departmento de Arquitectura de Computadores, Universidad Politécnica de Cataluña–Barcelona, December 1998.

[11] S. Palacharla, N. Jouppi, and J.E. Smith. Complexity-effective superscalar processors. *24th Annual International Symposium on Computer Architecture*, June 1996.

[12] C.D. Polychronopoulos. Nano-threads: Compiler driven multithreading. *4th International Workshop on Compilers for Parallel Computing*, November 1993.

[13] C.D. Polychronopoulos, M. Girkar, M.R. Haghighat, C.L. Lee, B. Leung, and D. Schouten. Parafrase–2: An environment for parallelizing, partitioning, and scheduling programs on multiprocessors. *International Journal of High Speed Computing*, 1989.

[14] M.A. Postiff, D. Greene, G. Tyson, and T. Mudge. The limits of instruction level parallelism in spec95 applications. *3rd Workshop on Interaction between Compilers and computer Architectures*, October 1998.

[15] E.M. Riseman and C.C. Foster. The inhibition of potential parallelism by conditional jumps. *IEEE Transactions on Computers*, 1984.

[16] E. Rotenberg, Q. Jacobson, Y. Sazeides, and J.E. Smith. Trace processors. *30th International Symposium on Microarchitecture*, December 1997.

[17] J.E. Smith. A study of branch prediction strategies. *8th Annual International Symposium on Computer Architecture*, 1981.

[18] G.S. Sohi, S.E. Breach, and T.N. Vijaykumar. Multiscalar processors. *22nd Annual International Symposium on Computer Architecture*, June 1995.

[19] J.G. Steffan and T.C. Mowry. The potential for using thread-level data speculation to facilitate automatic parallelization. *Fourth International Symposium on High-Performance Computer Architecture*, February 1998.

[20] G.S. Tjaden and M.J. Flynn. Detection and parallel execution of independent instructions. *Journal of the ACM*, October 1970.

[21] R.M. Tomasulo. An efficient algorithm for exploiting multiple arithmetic units. *IBM Journal of Research and Development*, January 1967.

[22] J.Y. Tsai and P.C. Yew. The superthreaded architecture: Thread pipelining with run-time data dependence checking and control speculation. *International Conference on Parallel Architectures and Compilation Techniques*, October 96.

[23] D.M. Tullsen, S.J. Eggers, J.S. Emer, H.M. Levy, J.L. Lo, and R.L. Stamm. Exploiting choice: Instruction fetch and issue on an implementable simultaneous multithreading processor. *22nd Annual International Symposium on Computer Architecture*, June 1995.

[24] S. Vajapeyam, T. Mitra, P.J. Joseph, and A. Mukherjee. Dynamic vectorization: The potential of exploiting repetitive control flow. Technical Report IISc-CSA-98-08, Dept. of Computer Science and Automation, Indian Institute of Science, August 1998.

[25] J.E. Veenstra and R.J. Fowler. Mint tutorial and user manual. Technical Report 452, Computer Science Department,The University of Rochester, June 1993.

[26] D.W. Wall. Limits of instruction-level parallelism. *4th International Conference on Architectural Support for Programming Languages and Operating Systems*, April 1991.

[27] W. Yamamoto and M. Nemirovsky. Increasing superscalar performance thorugh multistreaming. *International Conference on Parallel Architectures and Compilation Techniques*, October 95.

[28] T-Y. Yeh and Y.N. Patt. Alternative implementations of two–level adaptive branch predictors. *19th Annual International Symposium on Computer Architecture*, May 1992.