Quantifying the Benefits of SPECint Distant Parallelism in Simultaneous Multithreading Architectures

Daniel Ortega[†], Iván Martel[†], Venkata Krishnan[‡], Eduard Ayguadé[†] and Mateo Valero[†]

Alpha Development Group[‡]
Compaq Computer Corporation
Venkata.Krishnan@compaq.com

Departamento de Arquitectura de Computadores[†],
Universidad Politécnica de Cataluña – Barcelona, Spain
{dortega,imartel,eduard,mateo}@ac.upc.es

Abstract

In this paper we exploit the existence of distant parallelism that future compilers could detect and characterise its performance under simultaneous multithreading architectures. 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. We will show that distant parallelism can make feasible wider issue processors by providing more instructions from the distant threads, thus better exploiting the resources from the processor in the case of speeding up single integer applications. We also investigate the necessity of out–of–order processors in the presence of multiple threads of the same program. 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 main objective of compiler and processor designers is to effectively exploit the parallelism 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 parallelism that applications exhibit. However, the issue of where to look for this parallelism and how to exploit it is one of the main issues of today's microarchitecture research.

The current paradigm of computation for unithreaded applications is the superscalar out–of–order processor, where the parallelism is found among instructions of the unique control flow via the use of instruction windows. Several techniques can make this search of parallelism among instructions (ILP) more effective. Some of them, such as branch prediction strategies [17, 24], try to introduce more useful instructions into the processor window, allowing more instructions to be active at a time. Others such as register renaming [20] try to break false dependencies introduced by the compiler due to constraints in the number of logical registers. More recent techniques, such as data value speculation [9], try to benefit from the predictability of values in programs to break data dependencies. These techniques, which are just a sample, have as a primal objective the exposure of more data flows to the processor, thus allowing the exploitation of more parallelism among instructions.

Nevertheless, the parallelism found dynamically by these techniques has been shown to be bounded. Recent studies show that augmenting the instruction window size, thus increasing the dynamic scope of the processor may not be cost effective, not only because of technological reasons [12] but also because of the characteristics of the programs themselves, which show a saturation in the speed–up obtainable with highly aggressive architectures [15].

To overcome the limitations that a unique control thread may impose, the exploitation of multiple flows of control has been proposed [4, 23]. These multiple threads of control can be independent one from the other, thus simplifying the complexity of the processor. Nonetheless, automatically finding and exposing threads represents a complex problem. Recent hardware techniques have been proposed in which the hardware extracts multiple threads from a unithreaded application. These threads are speculative, and methods to detect mis–speculation and to recover from it have to be supported. Some hardware thread creation techniques propose to detect dynamic traces and/or speculate parallelism among them [16, 19]. Other techniques detect higher levels of semantics in the application and try to benefit from them: the detection of loops and the speculation of their iterations [10], the speculation of data dependencies between a loop and its continuation [21], or even the speculation of parallelism between a procedure call and its continuation [1].

One of the problems that impose all these hardware mechanisms that exploit thread level speculation is the enormous technological complexity they require. Detecting threads and recovering from misspeculation implies a lot of hardware that could be devoted to other computational tasks if threads could be statically explicited by the compiler. This reasoning is not new at all, the Multiscalar [18] architecture considered the exposure of threads as part of the compilers task.

Compilers have successfully exploited thread level parallelism in regular control structures such as low level loops, as it is done by POLARIS [2] or SUIF [6]. Even non–structured parallelism can be automatically detected when accurately combining the analysis of control and data dependencies in a hierarchical task graph [13] (like in the Parafrase–2 [14] or PROMIS compilers [3]).

All these efforts have redounded in an effective parallelising technology for numerical applications. However, integer applications are considered non–parallelisable because of the data and computation structures used in them. These applications tend to use dynamically allocated data structures (such as lists and trees) accessed through one or several levels of indirections, thus complicating the task of the compiler, which usually need the help of programmers by means of directives and assertions, multithreading libraries, or restructuration of the source code in order to expose parallelism. Because of all this, non–numerical applications are considered to be single threaded and little is expected from exploiting its parallelism.

In this paper we will show that non–numerical applications have inherent parallelism, and that reasonable performance gains can be expected from exploiting it. During the analysis of different non–numerical applications, we have found that they posses lots of semantic thread level parallelism [11], i.e. zones of code representing different computations which not necessarily must be done in a sequential order. However, semantic parallelism is difficult to find automatically, for many times the programmer or even the compiler may introduce dependencies among these parallel zones in the process of expressing the computation in a particular language. These false data dependencies must be therefore categorised and broken in order to expose parallelism. The techniques used to do so are not new, but those found in the parallelisation of numerical applications. The only difference is the non–homogeneous nature of the computation in non–numerical applications which may obscure the presence of parallelism.

The first objective of this paper is to show where to find this non–speculative parallelism and what techniques can be used to exploit it. The non–speculative nature of the parallelism we are aiming at will make the hardware more simple, therefore leaving room for other architectural enhancements. The second objective of the paper is to measure the benefits that this type of parallelism has under simultaneous multithreading architectures. For the programs evaluated, the parallel versions show a speed–up of up to 2.20 for a twelve way and up to 1.91 for an eight way out–of–order issue processor.

The organisation of the paper is as follows. Section 2 relates distant parallelism in integer applications to the already known types of parallelism in numerical applications. Section 3 summarises the compiler requirements that a future compiler should have in order to detect this kind of parallelism. Section 4 describes the simulation environment and the benchmarks analysed. A thorough analysis of results and their implications is done in Section 5. The paper ends with the conclusions in Section 6.

2. Distant Parallelism in Non–Numerical Applications

The main characteristic of distant parallelism is that, by definition, can not be detected at run–time, i.e. we have chosen not to search any type of thread level parallelism which could be detected by hardware mechanisms. This is implied by the term distant, which refers not only to dynamic distance, the number of dynamic instructions between two points in the execution of the program, but also to static distance, which is distance between two instructions in the code. Some hardware mechanisms exploit small distances in static code, such as speculating the continuation of loops or procedures, but our techniques ignore this type of parallelism, what makes them orthogonal to any of these hardware mechanisms, and also orthogonal to any other ILP oriented mechanism.

In the search for distant parallelism we have concentrated on zones of code defined by coarser loops. When looking at the relationship of dynamically executed instructions and static instructions in the code throughout execution time, one can see that certain zones of static code, coarser loops, are accessed in a repetitive way continuously, that is, the zone in particular is accessed during several *iterations* without intervening other zones of code. The pattern of access between iterations may not be exactly the same, especially when speaking of loops that cover bigger portions of static code. In fact, the bigger the loop, the higher probability of finding different paths of execution inside it. The loop is the natural way of simplifying how we express computation, and therefore, we have focused our search of parallelism in the computation they represent, either between iterations or inside each one.

This search of parallelism in loops is not new at all. Numerical applications have exploited the so called loop level parallelism for a long time, especially in low level loops. Nevertheless, the parallelism found in the lowest loop level in non–numerical applications is usually very poor. Low level loops in non–numerical applications are usually of

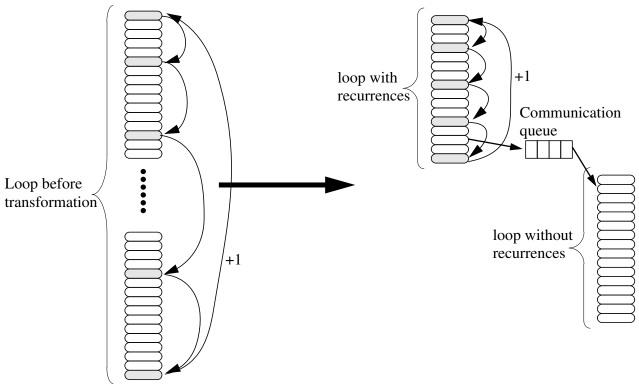

Figure 1. Decoupled asynchronous execution of a loop in `compress`

very few iterations, each one having little work to do and with many recurrences between them. Therefore, this low level of loops is not a good place to find parallelism in non–numerical applications.

At the code level, coarser loops can be seen as ways of expressing repetitive tasks that cover hundreds or thousands of instructions. To simplify the understanding of the computation, the programmer usually creates a static calling graph which responds to the algorithmic solution of the problem. All these function calls complicate the task of the compiler. Once understood the particular semantic problem the loop is addressing, the structure of the parallelisation becomes clear. The different techniques used are analogous to the ones used in non–numerical applications.

If the loop has a recurrence in it, we have tried to break the recurrence whenever possible. If this is not possible, as in `compress`, we have moved the instruction causing the recurrence up the instruction stream, and applied loop distribution to break the body of the loop in two zones, the one independent of the recurrence and the one dependent on

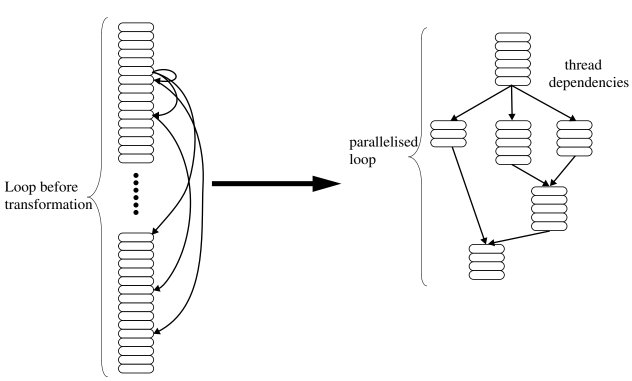

Figure 2. Thread scheduling in `m88ksim`

it. Once this is done, it is easy to execute both in parallel in a decoupled asynchronous way, with communication streams going from the first to the second one. This can be seen in Figure 1.

Sometimes, even data level parallelism can be found in non–numerical applications, especially those found in the field of multimedia processing, such as `ijpeg`. In this application, the loops were parallelised by splitting the work to be done among all the processors. These loops, although completely parallel, have plenty of function calls which do the processing.

Some coarser loops do not have this kind of parallelism between iterations. Nevertheless, if the loop is big enough and covers a large portion of static code, the probability of finding semantic parallelism in it is very high. An example of it is `m88ksim` or the colour transforming phases of `ijpeg`. In particular, in `m88ksim`, the coarser loop is the

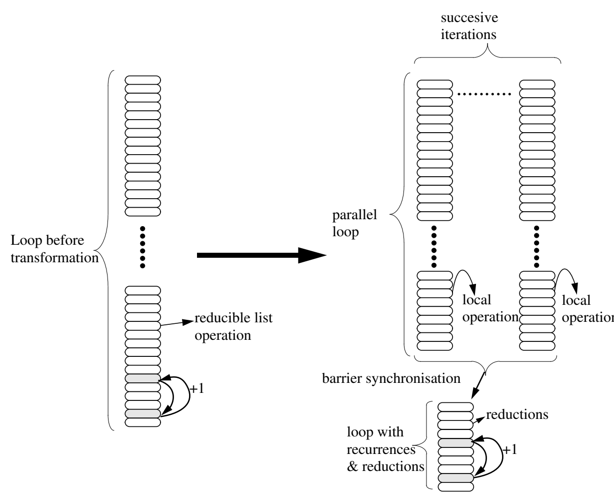

Figure 3. Loop splitting in `go`

one in charge of executing the instructions of the simulator. This execution implies a set of semantically parallel computations which cover enough code to create a thread for each of them (Figure 2). These threads were statically scheduled in parallel with synchronisations among them. If we were to make an analogy with the parallelisation of numerical applications, this task level parallelism would relate to the instruction scheduling done by current compilers in order to exploit the characteristics of superscalar out–of–order processors.

Finally, we have found loops that comprise lots of computation and that have dependencies among successive iterations, such as in `go`. In these cases, we have tried to isolate the dependencies and split the code in the loop. Some dependencies could be finally changed into a reduction operation, while others had to be separated from the rest of the loop in order to parallelise it. With code splitting we can leave the sequential execution of the iterations carrying the dependencies in one loop (Figure 3), and have a completely parallel loop with the rest of the computation found in the original loop. This technique is analogous to what has been called partial parallelisation in the field of numerical applications.

3. Compiler requirements

The objective of the parallelisations presented in this work is to show the benefits of thread level parallelism in integer applications. Therefore, they were not focused on covering large amounts of code nor in the fully exploitation of the characteristics of the algorithm being parallelised. We were more interested in generalising the ideas that could lead to the automatisation of the proposals explained before. Accordingly we made no changes in the algorithms but those that could be automatised, such as normal parallelising techniques like privatisation.

In the following paragraphs we will try to explain the compiler requirements that a parallelising compiler should have in order to automatically detect and exploit this type of parallelism.

The biggest requirement the compiler should have is an accurate interprocedural analysis able to disambiguate memory references and efficiently derive alias information. Coarser loop level usually traverses various levels of functions, accessing different data structures, many times through pointers. Once the compiler has detected the loop to be parallelised, a lot of work must be done to analyse data dependencies in a particular iteration and between iterations. This will possibly imply the construction of a task graph combining control and data dependencies in the form of task precedences.

If a particular loop is selected for parallelisation, the appropriate technique should be applied in order to create threads. Some kind of loop categorisation is needed to differentiate the possibilities of parallelisation. We have explained four different schemes that give parallel threads, but any technique already used in numerical environments to expose parallelism is liable of providing threads. Any of these techniques is likely to require code movement among threads in order to expose parallelism or even to balance the amount of work of each thread. Code movement usually implies a mechanism to estimate execution costs, either through program profiling or static estimation.

Another technique likely to be needed is variable privatisation, which we have used extensively in all the parallelisations. Not only scalar variables needed privatisation, but also structured variables and linked structures such as lists. Therefore, a complex pointer analysis is required to automatically implement these parallelisations.

Another technique closely related to privatisation is reduction. The detection of reduction operations should also be considered among the requirements of the parallelising compiler. An important fact, is that not only simple operations such as add should be considered reducible, but many more complex operations (e.g. merging of ordered lists) also fall in this category.

4. Evaluation environment

Processor Configurations

In this paper we focus on the influence of some characteristics that define current microprocessors: the order of issue, the issue width and the sizes of the first level of cache. The rest of the characteristics have been chosen so that they do not seem inappropriate with respect to these parameters.

We have made simulations varying the order of issue from a classical in–order issue processor to an out–of–order processor. Both configurations were studied with different issue widths and quantity of resources, trying to establish the relation of the parameters with the performance results they provided. The different issue widths can be seen in Table 1. We assume an aggressive superscalar core for both the in–order issue and the out–of–order issue. It can fetch and retire up to n instructions each cycle. A 2 K–entry direct–mapped 2 level branch prediction table allows multiple branch predictions to be performed even when there are pending unresolved branches. All instructions take 1 cycle to complete, except: integer multiply and divide take 2 and 8 cycles respectively; floating–point multiply takes 2 cycles, while divide takes 4 (single precision) and 7 (double precision) cycles.

Issue Width	Number of Functional Units (int/ld-st/fp)	Entries in Instruction Window	Number of Renaming Registers (int/fp)
4	4/3/2	32	32/32
8	6/3/4	128	128/128
12	12/6/4	200	200/200

Table 1. Characteristics of the processor core.

Finally, we model the memory sub–system in great detail. Caches are non–blocking with full load–bypassing enabled. We assume a perfect I–cache for all our experiments and model only the D–cache. We have analysed 3 different configurations for the L1 cache, with sizes of 16, 32 and 64 Kbytes and another one assuming perfect memory. Hits in L1 cache, L2 cache and main memory take respectively 1, 6 and 26 cycles of time. When simulating a simultaneous multithreading processor we have supposed one single L1 cache shared among all the threads.

Simulation Approach

Our simulation environment is built on a MINT–based execution–driven simulator [8]. MINT [22] captures both application and library code execution and generates events by instrumenting binaries. Our back–end simulator is extremely detailed and performs a cycle–accurate simulation of the architectures and hardware support for speculation described. The synchronisation introduced by our parallelisation strategies assumes that all communication must traverse L1 cache.

Benchmarks

The four programs analysed, `compress95`, `ijpeg`, `go` and `m88ksim`, belong to the SPECint95 suite. All of them were hand parallelised using standard semaphore and thread creating system calls. In the following paragraphs we are going to explain briefly the amount of code parallelised, the average number of threads in each of the zones parallelised and the theoretical speed–ups obtainable without considering any problems in the simulations. For a deeper explanation on the parallelisations, please refer to [11].

`compress95` has been divided in two different benchmarks, that comprise the compression and the decompression phase. Both of them were tested with a normalised data input, that covers one cycle of compression, the amount of data between two cleanings of the hash table used to compress. This input exhibits the same behaviour as the whole standard input, and was preferred for simulation timing reasons. In the compression phase, the parallelised zones reside in functions *compress* and *output*. The first thread is the one in charge of comprising the input and the second one does the compacting of the compression codes produced by the former. In the decompression phase the parallelism is found in functions *getcode* and *decompress*, the former being in charge of decompacting the input and the latter of decompressing the different codes to produce the output. Their duty is analogous to the functions parallelised in the compression phase. A theoretical analysis was done for both, measuring the average length in instructions of each of the threads created in the parallelisation. Supposing that the critical path could be reduced to executing only the longest thread, this would lead us to theoretical speed–ups of 1.24 for the compression and 1.72 for the decompression. Nevertheless, the high variance in length of the threads and the fact that the threads themselves are very short, which makes synchronisation overheads bigger in proportion, diminish the potential results observed under simulation.

The zones of code parallelised in `ijpeg` belong mainly to the following functions: *rgb_ycc_convert*, *h2v2_merged_upsample*, *forward_DCT* and *jpeg_idct_islow*. The first two functions transform data between three dimensional colour spaces, thus having potentially a speed–up of three. The other zones allow a parametrisable amount of threads, for they posses data parallelism. We have calculated the theoretical speed–up of having three threads in the first two zones and eight threads in the parametrisable ones. 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. All the simulations were done with the test input, and the theoretical speed–up according the profile information from these runs reached 1.7. Using profile information from the standard input yields a potential speed–up of 2.04.

In program `go` we concentrated our parallelisation in two analogous zones in functions *bdead* and *findcaptured* which mainly contain a loop that calls *iscaptured*. With the compiler transformations explained in Sections 2 and 3 we were able of executing a variable amount of calls to *iscaptured* in parallel. Using profile information as in `ijpeg` and the average amount of threads executed in parallel, we can estimate a theoretical speed–up for `go` of 1.7.

`m88ksim` is a processor simulator. A loop in *go_exec* is the one in charge of simulating all the instructions of the particular input program by executing the different phases of the execution in the processor. The body of this loop is constituted mainly by a function called *Data_path*. It is in this function where we have found the parallel zones of code in charge of simulating the different phases of execution. A maximum number of four threads can be executing at a time. All the simulations were done with test input. Using profile information as in the other examples we have concluded a potential speed–up of 2.7.

5. Analysis of results

In this section of results we will present a thorough analysis of the benefits of the parallelisations. First of all we will present some global data that supports our main goal: demonstrating that distant parallelism in non–numerical applications can derive performance gains in terms of execution time. The first results that show this tendency are the ones presented in Figure 4.

This figure presents six different configurations per program analysed (compress95 has been divided in compression and decompression phase). All configurations assume a 32 Kbyte L1 Cache and out–of–order execution. The six different bars differ in the issue width (four, eight and twelve instructions per cycle) and the version of the particular program (sequential versus parallelised). All the speed–ups were normalised with respect to the sequential 4 issue configuration.

The natural tendency of the sequential versions of the programs is to show a saturation in the ratio performance/issue width. Going from an issue width of four instructions to one of eight or twelve instructions yields a poor performance increase. The possible benefits of issuing up to

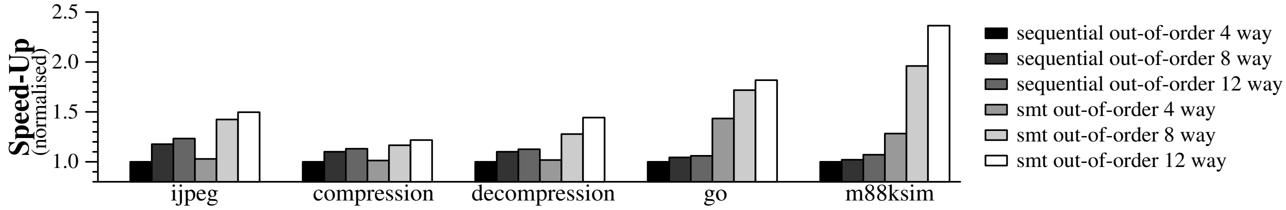

Figure 4. Normalised speed–ups of basic configurations for SPECint programs

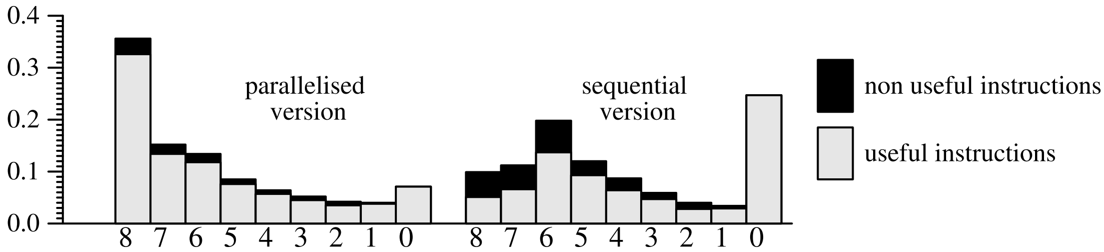

Figure 5. Normalised amount of cycles in which *n* slots could be issued for `m88ksim`

eight or twelve instructions per cycle, are diminished by the lack of ILP in this type of applications. The parallelised versions, however, exhibit a different behaviour. Although all the parallelised versions of the programs running under a four issue configuration go better than their respective sequential ones, and even sometimes better than the eight issue sequential ones, it is under the wider issue configurations where all the potential benefits derived from the parallelisation are shown.

We believe this tendency will be seen in any parallelised version of non–numerical applications. The exposure of thread level parallelism *provides* the processor with more useful instructions, overcoming the negative effect of low IPC (instructions per cycle). This can also be seen in Figure 5. We have chosen `m88ksim` to demonstrate this fact, although the rest of the applications under analysis show the same trend. We have analysed the number of cycles we were able of issuing *n* instructions in an eight way out–of–order processor with 32 Kbytes of Cache L1, both for the sequential version of the program and for the parallelised one. All the bars are normalised to the total amount of cycles of each simulation. The black shaded part of each bar represents the proportion of issued cycles that belong to instructions from paths following incorrectly branch speculated instructions. The first impression we get from this figure is that the histogram of the parallelised version exhibits a growing tendency with the number of instructions issued, reaching its peak in the maximum amount of instructions. The sequential version does not have this behaviour. The distribution of the sequential histogram is bimodal, having a peak in zero and in six instructions issued. Besides, this figure presents an interesting result. The proportion of non useful instructions is much less in the parallelised version. This is due to the exposure of more non speculative basic blocks, one for each thread, thus limiting the need of issuing speculative instructions.

Our second goal in this section is to show the performance gains that in–order issue processors have in presence of thread level parallelism, with respect to out–of–order superscalar architectures. To show this we have compared (Figure 6) a superscalar out–of–order, four, eight and twelve issue processor for the sequential versions, versus an in–order processor for the parallelised versions. All bars are normalised with respect to the sequential out–of–order four issue configuration. All the simulations of this figure assume a 32 Kbytes L1 cache, as in the previous figures.

The results in Figure 6 show that a parallelised version of a non–numerical application can exploit more parallelism in an in–order processor than what a sequential version of the same program can do with an out–of–order superscalar processor. With the exception of the compression phase of `compress95`, the rest of the benchmarks exhibit speed–ups when comparing the parallelised in–order simulations with the sequential out–of–order processors. The compression phase does not exhibit this behaviour because of the synchronisation overheads introduced in the compilation.

As happened with the out–of–order configurations, our threaded versions do not exhibit a great speed–up when running in four issue processors. When wider issue processors are compared, the comparison results are better and an outstanding conclusion can be made: in–order simultaneous multithreading processors can achieve better overall performance than out–of–order superscalar processors. It is not an intention of this paper to quantify the speed–up achievable in the low level process design due to an increase in clock rate, but we believe, partially from the papers published in

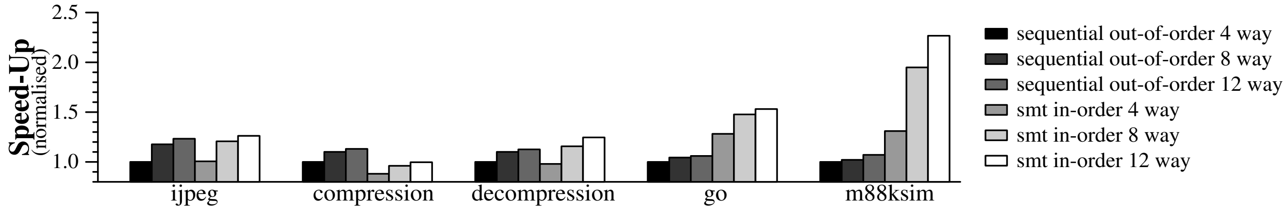

Figure 6. Normalised speed–ups of basic configurations for SPECint programs

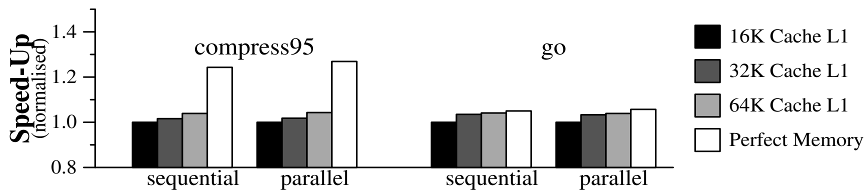

Figure 7. Normalised execution times of basic memory configurations for `compress95` **and** `go`

this field [12, 7], that introducing light weight threads in an in–order processor may be less costly in terms of cycle time than augmenting the instruction window and issue width. If this could be stated, we could affirm that an extra benefit could be expected for in–order multithreaded architectures.

We have also analysed the effect that our threaded versions may have in the cache hierarchy. We have seen no difference in behaviour between the sequential versions and the parallelised ones. The character of the applications in the cache hierarchy does not seem to be affected by the multithreaded versions of our programs, neither positively nor negatively. This effect can be seen in Figure 7. We have chosen two programs of our set of benchmarks, the compression part of `compress95` and `go` because they were representative of a high miss rate and a low one, respectively. All the simulations were run in an eight way, out–of–order processor with various cache sizes, as well as with perfect memory. Each version of the programs presents normalised speed–ups with respect to the worse simulation of the four, that of 16 Kbytes of L1 cache. We have normalised separately the sequential and the parallelised versions of both programs to be able of analysing the effect of the memory hierarchy alone. In this figure we can see that both parallel versions behave as the sequential counterparts. Instead of thinking that our parallelised versions behave exactly the same than the sequential ones, we believe that the way this kind of parallelism is expressed tends to counteract the benefits it may produce from prefetching or from memory latency tolerance with the disadvantages of having many threads polluting the cache. In future analyses we will try to investigate how to separate these two different effects, and the potential improvements it may derive.

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 [5]. Although several previous proposals have focused on the dynamic detection of these threads, 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.

Based in the existence of distant thread level parallelism demonstrated earlier in our research, we have focused in this paper in the measurement of the possible real benefits of this new kind of parallelism under simultaneous multithreading architectures. We have studied the possible effects of this type of parallelism with wider issue architectures than the ones currently in the market. We have also investigated the need for out–of–order architectures in the presence of multiple threads. Finally, we have analysed the effects that multiple threads from a single non–numerical application may have with respect to the size of the first level of cache.

Globally, we have shown performance speed–ups ranging from a 10% improvement to over doubling (2.20 speedup) the speed of the application on a twelve issue out–of-order machine, which taking into account the coverage of our hand parallelisations yields a very good impression on the possibilities of distant parallelism.

We can conclude that distant thread level parallelism imposes a new insight in the field of multithreading processors, making this type of architectures ideal for the task of increasing performance on non–numerical applications. This is due to the fact that the exposure of new types of parallelism in non–numerical applications can use more efficiently the resources of wide issue architectures.

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] H. Akkary and M. Driscoll. A dynamic multithreaded processor. *International Symposium on Microarchitecture*, 1998.

[2] 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.

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

[4] S. Eggers, J. Emer, H. Levy, J. Lo, R. Stamm, and D. Tullsen. Simultaneous multithreading: A platform for next generation processors. *IEEE Micro*, September/October 1997.

[5] A. Farcy and O. Temam. Improving single-process performance with multithreaded processors. *International Conference on Supercomputing*, May 1996.

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

[7] S. Hily and A. Seznec. Out-of-order execution may not be cost effective on processors featuring simultaneous multithreading. *International Symposium on High-Performance Computer Architecture*, January 1999.

[8] V. Krishnan and J. Torrellas. A direct execution framework for fast and accurate simulation of superscalar processors. *International Conference on Parallel Architectures and Compilation Techniques*, October 1998.

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

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

[11] I. Martel, D. Ortega, E. Ayguadé, and M. Valero. Increasing effective ipc by exploiting distant parallelism. *International Conference on Supercomputing*, June 1999.

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

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

[14] C. Polychronopoulos, M. Girkar, M. Haghighat, C. 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.

[15] M. 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.

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

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

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

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

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

[21] S. Vajapeyam, P.J.Joseph, and T. Mitra. Dynamic vectorization: A mechanism for exploiting far-flung ilp in ordinary programs. *International Symposium on Computer Architecture*, 1999.

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

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

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