Improving Single-Process Performance with Multithreaded Processors

Alexandre Farcy, Olivier Temam

Université de Versailles
45 avenue des Etats-Unis
78000 Versailles, France
{farcy,temam}@prism.uvsq.fr

Abstract

Multithreaded processors are an attractive alternative to superscalar processors. Their ability to handle multiple threads simultaneously is likely to result in higher instruction throughput and in better utilization of functional units, though they can still benefit from many (hardware and software) functionalities of superscalar processors and thus consist in an evolution rather than a radical transformation of current processors. However, to date, multithreaded processors have been mostly shown capable of improving the performance of multiple-process workloads, i.e. threads with independent contexts, but in order to compete with superscalar processors, they must also prove their ability to improve single-process performance.

In this article, it is proposed to improve single-process performance by simply parallelizing a process over several threads sharing the same context, using automatic parallelization techniques already available for multiprocessors. The purpose of this article is to analyze the impact of shared-context workloads on both *processor architecture* and *processor performance*. On a first hand, based on previous research works and by reusing many components of superscalar processors, a multithreaded processor architecture is defined. Then, considering the issues raised by shared-context workloads on data cache architectures have mostly been ignored up to now, we attempt to determine how current cache architectures can evolve to cope with shared-context workloads. The impact of shared workloads on this architecture is analyzed in details, showing that, like multiprocessors, multithreaded processors exhibit performance bottlenecks of their own that limit single-process speedups.

Keywords: Multithreaded processors, single-process performance, cache memories.

1 Introduction

Superscalar processors capable of issuing about 4 instructions per cycle are now becoming a standard. However, studies like [17] indicate that the amount of intrinsic instruction-level parallelism is limited. Besides, the necessity to concurrently execute a high number of instructions from the same instruction flow induces significant hardware overheads that increase with the degree of parallelism and the instruction window size from which parallelism is extracted, like for instance, the necessity to check dependences between a high number of instructions, registers that need to be dynamically renamed, buffering techniques to allow out-of-order execution.

Because of these hardware and software bottlenecks, it is not obvious that processor manufacturers will be indefinitely capable of scaling up superscalar processors. Consequently, several alternatives have been proposed in literature and industry. Some solutions require a deep modification of the processor execution model like data-flow architectures [24], while other solutions mostly consist in *evolutions* of current superscalar designs though they still address main performance bottlenecks. Considering the pace at which hardware processor architecture can sometimes evolve, less radical transitions are more likely to be contemplated by processor manufacturers. Three such "evolutions" are currently being considered.

The first solution is VLIW processors which has the advantage of strongly reducing hardware complexity but further increases the burden on the compiler for both detecting parallelism and efficiently scheduling instructions to functional units. The two other solutions, on-chip multiprocessors and multithreaded processors, have some similarities. The architecture of on-chip multiprocessors is simple but rigid: the threads scheduled on one processor cannot exploit the functional units of others so that the functional unit usage ratio may not be much higher than in current processors. Multithreaded processors, as described in the seminal article by Hirata [13], trade simplicity for efficiency: functional units are shared by the different threads each assigned to a logical processor. Because each logical processor needs to communicate with each functional unit, the hardware interconnection overhead for functional units and register banks can be high. On the other hand, functional units are likely to be more efficiently used.

This advantage of multithreaded processors over superscalar processors has been recently demonstrated in [30] by Tullsen, showing that multithreaded processors can be twice as efficient as high-degree superscalar processors (comparing 8 threads with 8 issue slots). However, up to now, the efficiency of multithreaded processors has been mostly demonstrated for heterogeneous workloads, i.e., when the workload is composed of distinct processes (8 in [30]). But, such architectures may prove viable solutions only if they are also efficient at increasing single-process performance. Multiple-instruction issue is suggested in [30] for each thread, in order to optimize single-process performance while decreasing global workload execution time using multithreading. With respect to single-process performance, this solution is efficient but it has the same limitations as superscalar processors (intrinsic instruction-level parallelism).

Multithreaded processor architectures allow coarser grains of parallelism to be used. The same techniques used for mul-

tiprocessors, i.e., parallelizing at the loop nest level, can be used. Thus, multithreaded processors can readily benefit from the large amount of research on automatic parallelization. Furthermore, the cost of dispatching blocks of iterations to all processors, which is a sequential and thus often costly process repeated prior to each parallel loop, is considerably reduced since logical processors are located on the same chip. Multiprocessors can exhibit limited efficiency because the granularity of parallelism within a loop nest is too small with respect to the number of processors and the initiation time. This obstacle disappears in multithreaded processors because the initiation time is fairly small, and most of all, because these architectures can tolerate small granularities (a single iteration). Consequently, the same parallelization techniques used in multiprocessors can be applied to multithreaded processors without the usually associated flaws. On the other hand, it is not obvious that new performance bottlenecks, specific to multithreaded processors, would not occur.

The purpose of this article is twofold: first to evaluate the capacity of multithreaded processors to improve single-process performance using the parallelization techniques that already exist for multiprocessors, and to evaluate the flexibility of multithreaded processors with different types of workloads. Second, and more important, the behavior of parallelized codes on multithreaded processors are examined in details. The influence of each component of the architecture is discussed and main performance bottlenecks are underlined. Techniques for coping with these limitations in future multithreaded processor architectures are discussed.

The multithreaded processor and cache architectures used are detailed in section 3. In section 4, the different aspects of the experimental framework (parallelization, trace collection, simulation) are presented. Finally, in section 5, multithreaded processors performance is analyzed.

2 Related Work

Up to now, several different concepts of multithreaded processors have been proposed. Some are simple modifications of current processors and have been actually implemented, others are more different concepts that still retain a strong resemblance with current processors. It is not the purpose of this article to propose a novel processor architecture: on the contrary, to make the results of this study relevant to a wide range of multithreaded processor architectures, specific innovations like thread synchronization support [13] or *context groups* [11], were not implemented. As mentioned above, the goal is to evaluate performance tradeoffs of multithreaded processor architectures, seen as an evolution of current superscalar processors. Still, the evaluation of parallelized processes and associated coherence issues made it necessary to propose possible evolutions of current data cache architectures.

Processor Architecture The concepts developed in the first multithreaded processor architectures were closer to data-flow machines like [24] than to superscalar processors. The goal was to both hide latency and increase instruction throughput by implementing enhanced context switching techniques (like in [2]); also novel thread interleaving techniques have been proposed [18, 9].

Because cache hierarchies, as used in current superscalar processors [1], are relatively successful at hiding latency, larger gains can be expected from the high instruction throughput capacities of multithreaded processors than from fast context switching techniques, and thus, recent studies are rather focused on instruction throughput issues.

In [30], a multithreaded processor based on the DEC 21164 is proposed with the goal of implementing *Simultaneous Multithreading*, i.e., several threads are run concurrently. Like in a superscalar processor, each thread is capable of issuing multiple instructions in the same cycle. In [11], the multithreaded processor architecture proposed is also close to a superscalar processor, but functional unit utilization is improved by grouping threads so that at least one thread per group is capable of issuing. In [13] like in the two previous studies, each thread can use any of the functional units. Each thread is assigned to one of the logical processors (1 up to 8 in [13]). Data dependences are resolved with a *scoreboard* technique [12], and resource dependences with stand-by stations implemented before each functional unit, thus limiting the impact of resource hazards on thread issue rate. Still, hardware support for *switch-on-cache-miss* is proposed, because this processor is to be used in a multiprocessor environment[2] where long latencies can be expected. Also, shared registers are used for inter-thread communications and also serve synchronization means. Single-process performance improvement is briefly discussed in this article: a 5.8 speedup with respect to a conventional RISC processor is reported using 8 threads and two load/store units (note that a perfect cache is used).

Sohi [27] and Franklin [7] also proposed a more novel architecture called *multiscalar processor*, between a multiprocessor concept and a multithreaded processor. A multiscalar processor allows parallel execution of tasks each on a distinct "execution unit". Those units share a common memory and communicate through a ring to forward register values. Unlike most multithreaded architectures mentioned above, functional units cannot be shared by the tasks. However, a multiscalar processor requires much work to be done at compile time to extract tasks and optimize scheduling, and hardware recovery mechanisms to allow speculative execution of tasks. In these studies, the possibility to improve single-process performance is evaluated [27], showing speedups up to 6 with an 8-unit multiscalar processor over a conventional processor on SPECint92, and the data cache issues of processors running multiple shared-context threads are examined in details [7].

Cache Architecture When multiple threads are run simultaneously, the number of cache accesses per cycle can be very high. With respect to instruction caches, the most simple solution is to use one cache per thread. The same solution can be considered for data caches. However, if a single process is parallelized over multiple threads, the same context is to be shared by several threads. The main constraint is then to maintain coherence. Assume one cache is used for each thread. *False-sharing* can happen (like in multiprocessors) in a parallel section with no dependence between tasks (iterations). Unless the lines that are not up to date are flushed at the end of the parallel section, coherence is not enforced. Because it can be very difficult to keep track of such cases (see [10]), it is not possible to extend the private-cache solution to a multithreaded processor where a process can be parallelized, i.e., where several threads can share the same context. This cache coherence problem is very similar to that of multiprocessors. However, unlike multiprocessors, this problem can be simply solved by having all threads that

[1] First-level and second-level cache on-chip in DEC 21164 [6], second-level on the processor module in Intel P6 [21].

[2] This processor was designed for an image synthesis application: the simulation of virtual worlds.

share the same context use the same cache space.

The issue is then to allow multiple accesses per cycle to this cache space. On a smaller scale, the same issue has already been raised for superscalar processors. Up to now, different forms of multi-banking and multi-porting have been used. Two-banked caches (odd/even) are used in the MIPS TFP [14] and T5 [23], Intel Pentium [15] and HP PA-8000 [20], allowing two simultaneous accesses to two different cache banks. Data replication is used in the DEC 21164 (two identical banks: stores are serialized to preserve coherence) and virtual multi-porting in the Power2 [19] (the SRAM access time is half the processor clock) allows two cache accesses in a single processor clock cycle. True multi-porting has been used in the Intel Pentium's [15] and IBM Power2's D-TLBs, using true dual-ported memory.

Actually, among all these solutions, it seems that only the multi-bank concept can scale up with the number of simultaneous cache accesses per cycle. For n ports, an SRAM with a clock $\frac{1}{n}^{th}$ the processor clock is unlikely if n is large. With respect to bank replication, copying the cache n times (without any effective gain in available cache space) is prohibitive. True multi-porting seems, at first, the most natural solution. However, the number of ports per SRAM increases the cache surface. Moreover, the energy required[3] to reload all capacity cells makes the SRAM cell access time longer [26]. Therefore, though n-port SRAMs, with n large, can be used in theory, the associated increased on-chip space and cache access time make it a prohibitive solution.

3 Multithreaded Processor Architecture

The main asset of the designs proposed by Hirata [13] and Tullsen [30] is their scalability, i.e., the number of functional units can be easily improved. For the sake of flexibility, a processor model which is inspired from Hirata's has been used in this study (see Figure 1) with logical processors issuing each a single instruction each cycle, and stand-by stations before functional units, as introduced in [13].

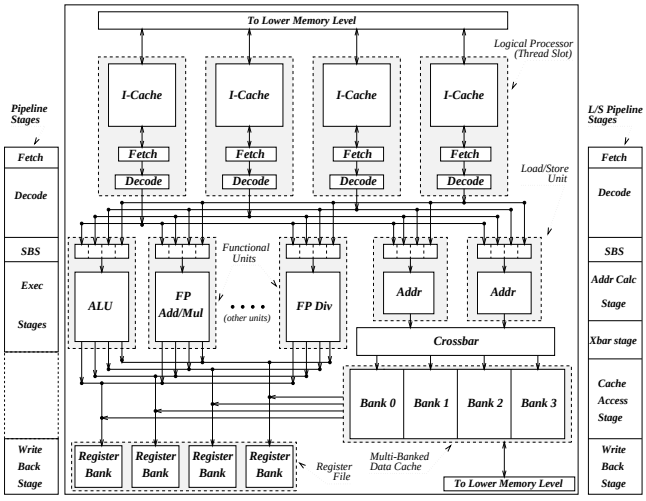

Figure 1: *Processor Architecture.*

However, unlike Hirata, register dependences are here resolved by having stand-by stations also play the role of reservation stations of a *Tomasulo* algorithm [12], allowing register by-passing. Like in [13], thread priority over each functional unit is managed with a simple round-robin algorithm, to avoid starvation.

[3] 1 port: 3.3mW/MHz, 2 ports: 2×3.5mW/MHz, 4 ports: 4×3.7mW/MHz.

The data cache used in our architecture is split into several independent banks, demanding a crossbar to distribute the requests from the load/store units to any bank. The main drawback of such a crossbar is the increased cache access time. Yet, a study [31] shows that passing through a 8×8 crossbar costs about 2 ns for a 32-bit path width. Other utilizations of crossbars in on-chip and time constrained environments, like the MIPS R8000 [22] crossbar to dispatch 4 instructions to the floating-point or integer units, and the full crossbar between the 5 ports of the Intel Pentium Pro's reservation station [21], further confirm the feasibility of such designs. With respect to performance, the main drawback of using a crossbar to access several banks is *bank conflicts*: two load/store instructions willing to simultaneously access the same bank must be serialized, thus stalling one of the load/store pipelines. The way a code is parallelized can strongly affect the probability of bank conflicts (see section 4).

In our implementation, each bank is either private to a thread, or can be gathered with several other banks to form a larger cache space that can be accessed by several threads belonging to the same context. Typically, n such threads share a cache space spread over n banks. With respect to bank sharing, only two radically opposite solutions are tested in [30]: either each process has a private cache or all processes share all cache banks. When the number of threads is equal to the number of cache banks (like in our case), Tullsen *et al.* show that both solutions are equivalent. Here, we propose an intermediate solution: to have the n threads of a given process context share n cache banks. This solution is flexible. If the number of active threads is smaller than the number of logical processors, single-thread processes can be provided with more than one bank. Though for small cache sizes this solution performs mostly like the all-banks-shared alternative, as cache size increases, inter-process misses tend to be dominant over intra-process misses, as shown in [1]. Thus, in the future, it will be more critical to avoid inter-process misses by privatizing groups of banks to processes, than to limit intra-process capacity/conflict misses by further increasing the available cache space. More important, this intermediate solution reduces the occurrence of bank conflicts.

If multiple load/store units are used, further coherence issues also arise. Two instructions accessing the same address in two different load/store pipelines have to complete in-order. Assume one pipeline is stall (because of bank conflicts or any cache stall), it is necessary to stall the other load/store units to enforce in-order load/store execution, as no out-of-order execution of memory references mechanism is provided in our model.

The cache banks used in this study are non-blocking to decrease the impact of such stalls (with a *miss queue* close to the DEC 21164's [6]). Using multiple cache banks and a crossbar has also been suggested in [28] and [30] but the associated coherence issues were not considered in detail. However, Franklin and Sohi [7, 27] propose to use an ARB (Address Resolution Buffer) to handle the above mentioned coherence issues (memory disambiguation) in their multiscalar processor. The ARB is a cache placed between the crossbar and the cache bank which keeps track of the different load/store instructions accessing the same address. When an illegally-ordered load/store sequence has been detected, the corresponding task is re-executed from the conflicting instruction. Aside from its hardware overhead, this solution is difficult to extend to multithreaded processors. Indeed, it is based on the fact that, in a multiscalar pro-

cessor, tasks are separated in small sets of instructions (one or few basic blocks) dispatched in a rotating manner to the different execution units. Thus, once the oldest execution unit has completed, it is certain that no load/store order issue can arise. Since the size of tasks is small, few cache lines could potentially be victim of coherence conflicts at the same time so that the ARB needs not be large. In our case, with n logical processors, one thread can correspond to $\frac{1}{n}^{th}$ of a parallel loop, which is far too large for an ARB. Another possible solution is the *Address Reorder Buffer* [20] of the HP PA-8000's load/store unit which allows out-of-order execution of memory references and speculative execution of load/store instructions by buffering pending LOADs and STOREs and comparing the addresses of these requests to new load/store requests to detect coherence issues.

4 Experimental Framework

Parallelization To sustain the argument that current techniques on automatic parallelization can be readily applied to multithreaded processors, it was critical to use an automatic parallelizer currently available for multiprocessors. We have selected KAP, a commercial Fortran preprocessor designed by KAI. KAP was used to detect parallel loop nests and to mark these loops as *parallel* in the source-code. For that task, the efficiency of KAP was not high, but we expect to obtain better results in the future either with improved versions of KAP or with state-of-the-art academic parallelizers like Polaris [3] or SUIF [4] including more efficient data dependence analysis techniques (the Omega Test [25] was recently implemented in Polaris).

Because the goal was to measure the best performance that can be achieved with a parallelized process on a multithreaded processor, only loop nests without loop carried dependences were parallelized, i.e., no synchronization techniques were used. The outer loop of each parallel loop nest was parallelized. The two most classic techniques for statically distributing loop iterations over different processors (logical processors in our case) are by blocks of contiguous iterations, and by assigning iteration i to processor $i \ mod \ P$ where P is the number of logical processors. For cache-based systems like KSR-1, modulo scheduling may often behave poorly because many processors need to access the same cache line at the same time. In the cache design selected (see section 3), this results in cache bank conflicts. Therefore, a block processor scheduling technique was finally selected. Loops are simply parallelized as shown below:

```
DO I=1,N
  Loop Body
ENDDO
```

Original loop.

```
DOALL II=1,N,B
DO I=II,min(II+B,N)
  Loop Body
ENDDO
```

Parallel loop.

where $B = \lceil \frac{N}{P} \rceil$.

Traces and Simulation Object-code traces are collected with the Spa package [16]. The main difficulty is to collect parallel traces in such a way that different parallelization alternatives (block/modulo, varying number of logical processors) can be evaluated. The processor to which an iteration of a parallel loop must be assigned is determined on-the-fly during trace collection. The only informations required are a flag indicating the beginning/end of a parallel section and the iteration number of the loop being parallelized [4].

As explained above, parallel loops are detected with KAP and marked as such in the source-code. Then, the Fortran-to-Fortran compiler Sage++ [8] is used to instrument the code with markers (references to known addresses) so that the beginning/end of parallel loops and the beginning of each iteration can be detected during code execution. Then, Spa was modified to acknowledge these marker addresses and augment the trace with the desired informations.

A last delicate issue was the fate of scalar variables, like loop indices. If the same scalar variable is used by multiple threads, numerous bank conflicts are likely to occur. Thus, it is necessary to privatize scalar variables to each thread by shifting their addresses, so that two such variables don't fall in the same cache bank.

We developed our own simulator to test varying alternatives of multithreaded designs because few multithreaded simulators were available. Still, we contemplated using Concurro [11], but the specific features of this design (especially the principle of gathering threads in *context groups*) were too restrictive. As explained in section 3, the multithreaded processor architecture simulated is close to the one described in [13], except for the memory system parts which were not described in that article. The functional units latency of the processor architecture simulated are 1 cycle for *integer ALU*, 2 cycles for *floating-point Add/Multiply*, 16 cycles for *floating point Divide* and 3 cycles for *Load/Store*. Except for *FP Divide*, all functional units are pipelined. [5]

Each thread is assigned to a logical processor which includes the logic for *fetch* and *decode* stages. Each thread is assigned a private instruction cache bank and a private register bank. Each cache bank is 8-Kbyte large, direct-mapped with a 32-byte line and write-back. The number of data/instruction cache banks is always equal to the number of logical processors, i.e. the number of threads. The miss latency is equal to 15 cycles and 1 miss request can be issued every 2 cycles.

The codes were compiled on a Sparc-5 workstation.

Benchmarks In this study, both shared-context and a combination of shared-context and private-context workloads are executed. Thus, programs corresponding to classic processor workloads as well as benchmarks that lend to automatic parallelization techniques are needed. For the first category, we selected `Gcc`, `Compress`, `Simulator`, `Contour`, `Grep`, `Latex` and `Segmentation` [6]. These seven programs were only used to increase the load on the processor and no specific performance evaluation was done on those. For numerical codes, we picked three of the Perfect Club [5] benchmarks: `FLO52`, `ARC2D`, `MDG`. Since this study is focused on analyzing the behavior of multiple shared-context threads, statistics are collected over parallel sections only, even though all instructions are executed. Considering our goal was to evaluate the best achievable performance with a multithreaded processor, we chose the three codes on which KAP was most efficient at finding parallel loops (other codes had too short parallel sections). The fraction of instructions in parallel sections is 50% for `FLO52`, 63% for `ARC2D`and 16% for `MDG`. The miss ratio of each code on a single cache bank is shown in Figure 2.

[4] In fact, to apply block processor scheduling, the total number of iterations in a parallel loop must be known, thus two passes are necessary for each code.

[5] These values have been averaged over several current microprocessors (MIPS R8000, MIPS R10000, PowerPC 604).

[6] *Contour* and *Segmentation* are image analysis codes; *Simulator* is our processor simulator.

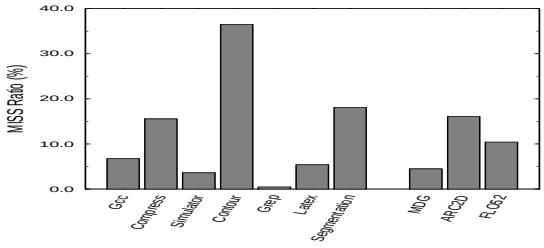

Figure 2: *Miss ratio of all benchmarks.*

5 Performance Evaluation

Because it is unlikely a large number of logical processors can be implemented on a single chip in the near future, the experiments have been limited to 8 logical processors in each case. For the sake of clarity, in all experiments the number of functional units of each class [7] is increased altogether, but detailed analysis of the impact of each functional unit is also provided. Thus a single number is used to characterize the number of functional units unless otherwise specified, "two functional units" meaning "two functional units of each class". In this section, various workloads are analyzed. First, global behavior is presented, then the different origins of performance bottlenecks are progressively detailed.

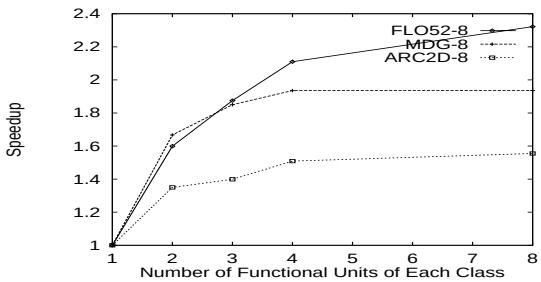

Figure 3: *Speedup with respect to one thread.*

Global behavior of parallelized processes In Figure 3, each program is parallelized into 8 threads (a program `PRG` parallelized into n threads is denoted `PRG-n`), and the number of functional units of each class is varied from 1 up to 8. It appears that beyond 4 functional units, i.e., 4 functional units of each class, only little performance improvements can be expected [8]. In fact, when n threads are run on a multithreaded processor, the number of functional units needs not be equal to n to achieve maximum performance thanks to the sharing of functional units by the different threads. This is actually one of the major arguments for supporting multithreaded processors: in terms of functional units, scaling up a multithreaded processor is cheaper than scaling up an on-chip multiprocessor. However, for `FLO52` and more particularly for `ARC2D`, there are obviously performance bottlenecks other than resource issues that limit performance improvements. This can be seen on Figures 4 and 5, where the number of threads is varied from 1 up to 8 with the number of functional units equal to 4 for each class (i.e., near best performance for 8 threads). The maximum speedup achieved by `FLO52` is about 5.5 and 4 for `ARC2D`, i.e., half the optimal speedup for 8 threads. As a result, the CPI remains relatively high (0.3 for `ARC2D` and 0.25 for `FLO52`) considering the number of threads. In other terms, multithreaded processors exhibit sub-linear speedups, like multiprocessors, even though they are not impaired by the same remote-communications and task distribution overhead issues, or like superscalar processors, even though the intrinsic parallelism is high.

[7] the 4 classes of functional units provided in our architecture are *integer ALU*, *FP adder/multiplier*, *FP divider* and *Load/Store*.

[8] the speedup is computed with respect to one thread run on a 4 functional units per class processor, in terms of CPI.

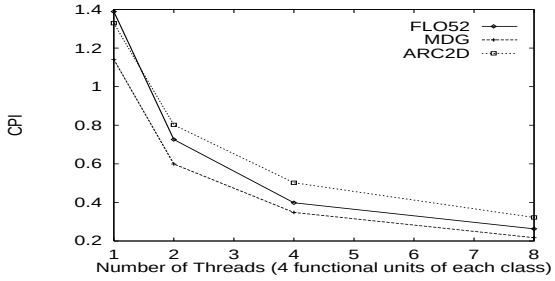

Figure 4: *CPI.*

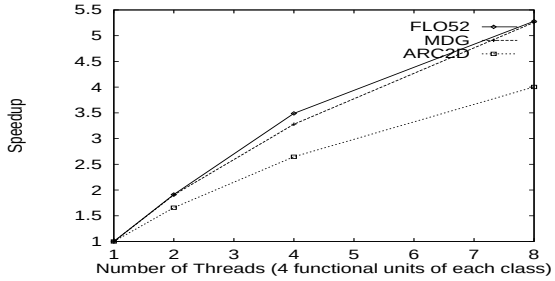

Figure 5: *Speedup with respect to one thread.*

Origins of stalls Six stall causes can be distinguished in Figure 6: WAW register dependences, branch delays, resource conflicts, busy/stall load/store units, instruction cache miss and thread synchronization. Let us examine in more details when each of these stalls can occur. A logical processor can be stall only if a stall occurs in one of the two stages directly controlled by the logical processor: the fetch and decode stages. Otherwise, as far as a logical processor can issue, no stall cycle is counted, even if some functional units are stall for a reason or another.

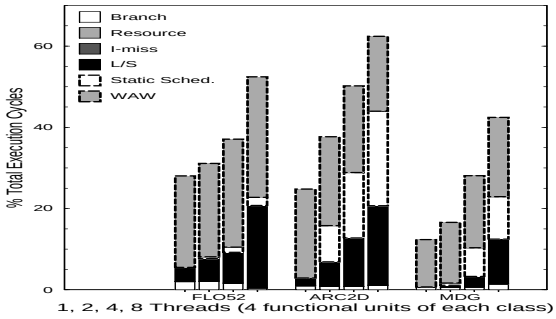

Figure 6: *Distribution of stall cycles.*

Consequently, with respect to *register dependences*, only Write After Write dependences can directly result in a stall cycle: the decode stage cannot issue an instruction if a functional unit is about to write in its destination register. In case there is a RAW (Read After Write) dependence but a stand-by station is available before one of the corresponding functional units, the instruction is issued and no stall cycle occurs. WAR (Write After Read) dependences cannot occur as registers are read and reserved in the decode stages in program order. As can be seen in Figure 6, WAW dependences account for a major share of stall cycles. Thus, dynamic register renaming techniques, as used in superscalar

processors can strongly benefit to multithreaded processors as well. On the other hand, the effect of WAW dependences is relatively stable as the number of threads increases, and thus other stall causes then seem to become significant if not dominant.

No sophisticated branch prediction technique has been used to avoid *branch delays* for the moment, but considering the codes parallelized are numerical codes, i.e., made of loops, the effect of branch penalties does not show much.

With respect to *resource conflicts*, the decode stage is stall if all corresponding functional units and their stand-by stations are busy (if all functional units are busy, the free stand-by stations can be used to hide resource conflicts as well). The distribution of resource conflicts is shown in Figure 7: integer and floating-point `add` and `mul` operations are responsible for most conflicts (the resource conflicts statistics in Figure 7 do not include load/store units). As can be seen, threshold performance can be achieved with 3 integer and floating-point `add/mul` functional units, and a single unit for each other operation. Note that the number of `add` units necessary is close to the optimal values of 3 integer `add` and 2 floating-point `add` found in [17] for an 8-issue superscalar processor.

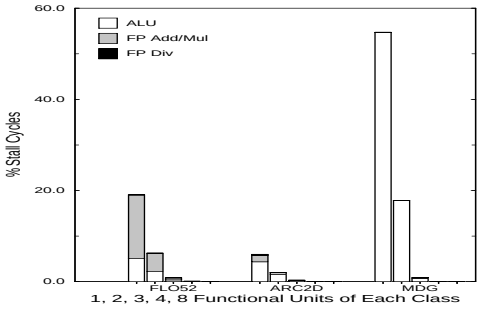

Figure 7: *Distribution of resource conflicts (fraction of stall cycles) as a function of the number of functional units.*

With respect to *load/store units*, there are multiple causes of stalls: resource conflicts, but also stalls due to cache accesses. As the number of threads increases, load/store-related stalls increase significantly (see Figure 6). Thus, such stalls are likely to be a major cause of speedup limitations. They will be examined in more details in the next paragraphs.

Finally, a consequence of static allocation of iterations on the logical processors (*static scheduling*) is that some threads may complete execution earlier than others depending on the distribution of iterations. In an actual implementation of a multithreaded processor, two solutions can be adopted: synchronizing threads (early threads wait for other threads to complete) or switching an active thread on the logical processor of an idle thread. Here, the first technique was used in order to highlight the effect of static processor scheduling. As shown in Figure 6, while this effect is negligible for `FLO52`, it becomes dominant for `ARC2D` when the number n of logical processors increases. The consequence of static processor scheduling, as applied here, is that one thread may receive up to $n-1$ less iterations than other threads for any parallel loop nest. This effect does not show in `FLO52` because, for many of its loops, the number of iterations (32) is a multiple of the number of logical processors. Other techniques like modulo processor scheduling, which allows a better distribution of iterations, behave even worse because many threads simultaneously access consecutive data, i.e., the same cache line, resulting in bank conflicts. Clearly, better static processor scheduling techniques or implementing dynamic allocation of iterations to logical processors should be considered. Still, this load balancing phenomenon would have a far stronger impact on a classic multiprocessor.

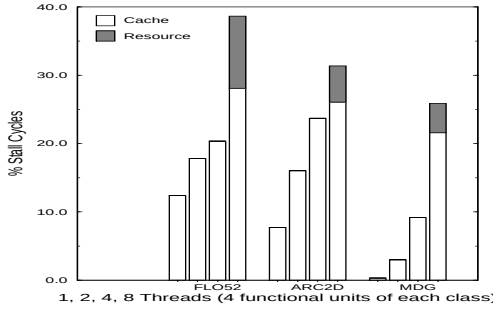

Figure 8: *Distribution of stall cycles due to load/store functional units.*

Stalls due to load/store instructions There are two main categories of stalls due to load/store instructions: cache stalls (for coherence or misses) and load/store unit resource conflicts. As can be seen in Figure 8, load/store resource conflicts only occur past 4 threads (recall there are 4 load/store units in the experiments of Figure 8). This phenomenon is specific to load/store units, mostly because coherence issues forbid using load/store units stand-by stations to hide resource conflicts. Indeed, once a memory instruction has been sent to the stand-by station of one of the load/store units, other LOADs or STOREs of the same thread cannot be sent to another stand-by station to avoid out-of-order execution of load/store instructions (see section 3).

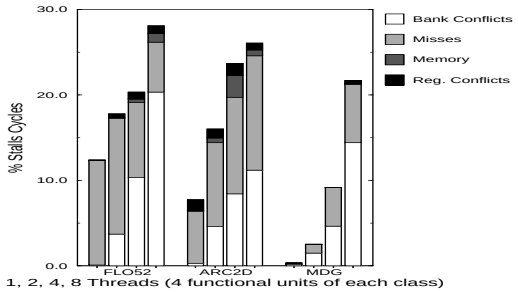

Figure 9: *Distribution of stall cycles due to cache stalls.*

Cache stalls The major cause of load/store stalls are cache stalls. Four kinds of cache stalls can be distinguished: register accesses, memory bandwidth, misses and bank conflicts (see Figure 9). *Register access conflicts* occur when two references (cache hit, line reload, reference leaving the miss queue...) need to access the write register port of a given thread at the same time. Though such conflicts exist and require special hardware to handle, their impact is often negligible. Because the first-level cache size increases with the number of threads, the increasing issue rate of load/store instructions does not seem to induce an excessive *memory* request issue rate (see Figure 9, graph `Memory`). This is true only because cache banks are write-back. Indeed, the performance of write-through cache banks is worse because the high amount of memory store requests induce numerous cache stalls (particularly for `FLO52`, see Figure 10). Still, the impact of multithreaded processors on lower memory levels should be examined in more details.

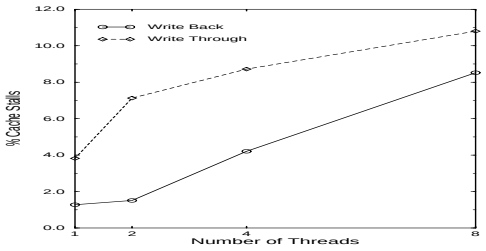

Figure 10: *Write-Through vs. Write-Back cache stalls due to memory contention (for* FLO52*).*

According to Figure 9, the two main causes of cache stalls are *misses* and *cache-bank conflicts*. Since cache banks are non-blocking, misses effectively induce cache stalls in two cases: to copy back a dirty line to write-buffer, and when a miss occurs on a line waiting for a pending miss request. When a thread is starved waiting for a load/store that missed, the decode stage carries on issuing instructions and sending them to stand-by stations, thus no stall occurs immediately. On the other hand, the thread must stall if all stand-by stations are busy (or one load/store stand-by station). Thus, the effect of memory latency also feels indirectly through resource conflicts (particularly on load/store stand-by stations). Copying a dirty line to a write buffer is a cheap operation (1 cycle) but it happens frequently. Also, to our surprise, misses on pending lines occur relatively often. This is related to the nature of parallel threads: it sometimes happens that several threads need to read the same line at the same time, since some array data can be shared by several iterations or the different data needed by several threads belong to the same cache line. Still, as the cache size increases, less than half of cache stalls are related to misses (see Figures 9 and 11), except for ARC2D. Thus, it seems that a combination of parallel threads and non-blocking cache banks are efficient at hiding small latencies. Consequently, switch-on-miss policies that require additional hardware support may not be necessary. With respect to ARC2D, a specific phenomenon occurs: in this code, there are numerous references with power-of-two strides which thus induce cache conflicts even for large cache sizes [29].

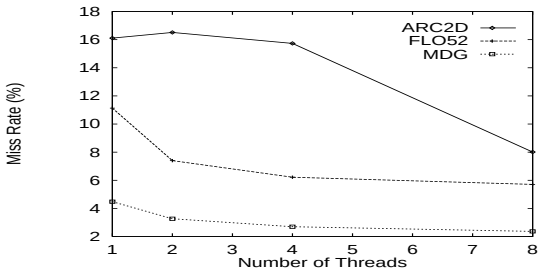

Figure 11: *Variation of process total miss ratio with the number of threads.*

On the other hand, for all three benchmarks, the effect of cache bank conflicts steadily increases with the number of threads. The impact of cache bank conflicts on total execution time is particularly strong because all load/store units must be stall to preserve coherence each time a conflict occurs (see section 3). The effect of bank conflicts is further worsened by the poor distribution of conflicts over cache banks, as shown in Figure 12 for 8 threads. Thus, more elaborate and selective coherence schemes that require only partial stalling would make the present cache architectures less sensitive to bank conflicts. The second complementary research direction is to investigate the possibility to apply existing solutions to memory-bank conflicts in vector processors to multiple-bank caches.

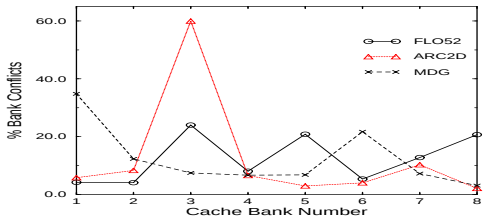

Figure 12: *Distribution of cache bank conflicts (8 threads).*

Heterogeneous Workloads While the above paragraphs dealt with single-process performance, multithreaded processors also need to prove flexible, i.e., to perform well with heterogeneous workloads (a combination of parallelized processes and private-context threads). In Figure 13, two experiments were conducted. Graph N + PRG-4 corresponds to a program parallelized over 4 threads (thus the allocated cache space remains constant) and n other private-context threads are run simultaneously.[9] This graph shows that running an increasing number of threads does not affect much the performance of the parallelized process. Still, the small performance degradation corresponds to additional memory traffic which further delays memory requests of the parallelized process. However, crossbar conflicts do not increase thanks to the pseudo-privatization of cache banks to processes. The purpose of graph N + PRG-(8-N) is different: to evaluate whether a multithreaded processor performs better within a heterogeneous environment (threads mostly have distinct contexts) or within a homogeneous environment (threads mostly share the same context). In graph N + PRG-(8-N), 8 threads are always run in total but the program is parallelized over $8 - n$ threads; other single-thread processes are also added according to the order of the list in section 4. It appears that a homogeneous workload performs better because of the cache structure adopted; however note that our cache design privileges n shared-context threads over n private-context threads, because in the first case, all threads share the n cache banks, while in the latter case each thread only receives a single bank, thus a smaller cache space. Still, the stiff difference is also due to the fact that the total working set of a homogeneous workload is often smaller than that of a heterogeneous workload since parallel threads usually share some data.

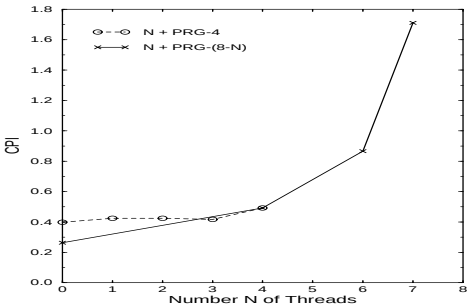

Figure 13: *Heterogeneous workloads.*

[9] The additional processes are *gcc* for 1 additional process, *gcc+compress* for 2 additional processes, and so on, according to the order of the list in section 4.

6 Conclusions and Further Work

Based on the experiments of this study, the conclusions on multithreaded processors are mitigated: potential capacities of such architectures are significant, but further hardware improvements are required to exploit them. Multithreaded processors can effectively improve single-process performance, and best achievable performance is reached with fewer functional units than the number of threads. On the other hand, the usual assumption that multithreaded processors can avoid the complexity of superscalar architectures is mostly wrong, since both architectures require similar hardware enhancements, like dynamic register renaming and memory disambiguation. More important, only sublinear speedups are obtained because cache bandwidth cannot be fully exploited, even though a cache design with sufficient potential bandwidth can be designed with current technology. The two main limitations are the necessity to preserve coherence which induce excessive cache stalls, and cache bank conflicts.

Thus, before superscalar processors can mutate into multithreaded processors capable of supporting multiple threads simultaneously, several issues must be investigated. Automatic parallelization techniques are not mature enough to avoid using multiple-issue threads, as proposed with *Simultaneous Multithreading* [30]. On the other hand, automatic parallelization can be used to increase intrinsic parallelism in many cases. So that single-process speedup does not decrease with the number of threads, a more selective cache coherence scheme and techniques for reducing cache bank conflicts must be developed. Also, the hardware cost related with implementing dynamic scheduling of tasks over shared-context threads would widen the application of multithreaded processors to codes with very few iterations per loop, i.e., that typically perform poorly on multiprocessors. The scope of multithreaded processors would be further enhanced if fast synchronization techniques are implemented, possibly through registers or caches, to parallelize loops with dependences.

References

[1] A. Agarwal, J. Hennessy, and M. Horowitz. Cache Performance of Operating Systems and Multiprogramming. *Transactions on Computer Systems*, pages 6(4):393–431, November 1988.

[2] Anant Agarwal, Beng-Hong Lim, David Kranz, and John Kubiatowicz. APRIL: A Processor Architecture for Multiprocessing. In *International Symposium on Computer Architecture*, pages 104–114, June 1990.

[3] B. Blume et al. Polaris: The Next Generation in Parallelizing Compilers. In *Proceedings of the Seventh Workshop on Languages and Compilers for Parallel Computing*, 1994.

[4] Computer Systems Laboratory, Departments of Electrical Engineering and Computer Science, Stanford University. *The Stanford SUIF Compiler*, 1994.

[5] George Cybenko, Lyle Kipp, Lynn Pointer, and David Kuck. Supercomputing Performance Evaluation and the Perfect Benchmarks. In *Supercomputing '90*, pages 254–266, 1990.

[6] Digital Equipment Corporation, Maynard, Massachussets. *Alpha 21164 Microprocessor, Hardware Reference Manual*, 1994.

[7] Manoj Franklin. *The Multiscalar Architecture*. PhD thesis, University of Wisconsin, Madison, 1993.

[8] D. Gannon et al. SIGMA II: A Tool Kit for Building Parallelizing Compiler and Performance Analysis Systems. Technical report, University of Indiana, 1992.

[9] R. Govindarajan, S.S. Nemawarkar, and Philip Lenir. Design and Performance Evaluation of a Multithreaded Architecture. In *International Symposium on Computer Architecture*, pages 298–307, 1995.

[10] Elana D. Granston and Harry A. G. Wijshoff. Managing Pages in Shared Virtual Memory Systems: Getting the Compiler into the Game. Technical Report 92-19, Department of Computer Science, Leiden University, December 1992. Submitted to *International Conference on Supercomputing*.

[11] Bernard Karl Gunther. *Superscalar performance in a Multithreaded Microprocessor*. PhD thesis, University of Tasmania, Hobart, December 1993.

[12] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., 1990.

[13] Hiroaki Hirata et al. An Elementary Processor Architecture with Simultaneous Instruction Issuing from Multiple Threads. In *International Symposium on Computer Architecture*, pages 136–145, 1992.

[14] Peter Yan-Tek Hsu. Designing the TFP Microprocessor. *IEEE Micro*, pages 23–33, April 1994.

[15] Intel Corporation. *Pentium Processor User's Manual*, 1993.

[16] G. Irlam. *SPA package*, 1991.

[17] Stephan Jourdan, Pascal Sainrat, and Daniel Litaize. Exploring Configurations of Functional Units in an Out-of-Order Superscalar Processor. In *International Symposium on Computer Architecture*, pages 117–125, July 1995.

[18] Daniel C. McCrackin. Eliminating Interlocks in Deeply Pipelined Processors by Delay Enforced Multistreaming. *IEEE Transactions on Computers*, 40(10):1125–1132, 1991.

[19] Microprocessor Report, vol.7, no 13. *IBM Regains Performance Lead with Power2*, October 1993.

[20] Microprocessor Report, vol.8, no 15. *PA-8000 Combines Complexity and Speed*, November 1994.

[21] Microprocessor Report, vol.9, no 17. *Intel Boosts Pentium Pro to 200 Mhz*, November 1995.

[22] MIPS Technologies Incorporated. *R8000 Microprocessor Chip Set, Product Overview*, 1994.

[23] MIPS Technologies Incorporated. *R10000 Microprocessor Chip Set, Product Overview*, 1994.

[24] R.S. Nikhil, G.M. Papadopoulos, and Arvind. *T: A Multithreaded Massively Parallel Architecture. In *International Symposium on Computer Architecture*, 1992.

[25] W. Pugh. The Omega Test: a fast and practical integer programming algorithm for dependence analysis. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 1992.

[26] Allan L. Sillburt et al. A 180-MHz 0.8-μm BiCMOS Modular Memory Family of DRAM and Multiport SRAM. *IEEE Journal of Solid-State Circuits*, 28(3):222–231, March 1993.

[27] Gurindar S. Sohi, Scott E. Breach, and T.N. Vijaykumar. Multiscalar Processors. In *International Symposium on Computer Architecture*, pages 414–425, July 1995.

[28] Gurindar S.Sohi and Manoj Franklin. High-Bandwidth Data Memory Systems for Superscalar Processors. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 53–62, April 1991.

[29] O. Temam, C. Fricker, and W. Jalby. Cache Interference Phenomena. In *Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, 1994.

[30] Dean M. Tullsen, Susan J. Eggers, and Henry M. Levy. Simultaneous Multithreading: Maximizing On-Chip Parallelism. In *International Symposium on Computer Architecture*, pages 392–403, 1995.

[31] B. Zerrouk, J.M. Blin, and A. Greiner. Encapsulating Networks and Routing. In *Proceedings of the 8th International Parallel Processing Symposium*, 1994.