Tipi: The Threads Processor

Bernard Goossens
LIAFA, Université Paris 7 Denis Diderot, 2 place Jussieu
75251 Paris cedex 05, France
tel: 33 1 44 27 54 43
fax: 33 1 44 27 68 49
email: bg@liafa.jussieu.fr

Abstract

Tipi is an implementation proposal of the SMT architecture. This implementation is very close to a multiprocessor architecture, sharing only the second level caches and the functional units. It is designed to achieve a high performance with both a high IPC and a very short cycle (estimated to 10 gates levels). Functional unit sharing and very short cycle are essential to allow scalability by avoiding unnecessary copies of unfrequently used operators like dividers or floating points units. High IPC is obtained by scalable logic allowing a high superscalar degree (the peak issue rate is a full instruction cache block per cycle) and a high number of threads (the 10 gates levels critical path is based on an 8 threads design; the parts of the logic relying on threads number t have a gates levels complexity proportional to $log_2 t$, so that doubling t should not affect cycle much). Very short cycle comes from a careful implementation of the Tomasulo's algorithm, using a future file rather than register renaming and assuming a static or dynamic allocation of operators. The caches and the arithmetic operators are also constrained to fit in their pipe stage.

Keywords:

on-chip multiprocessor, very short cycle pipeline, SMT, multithreading, superscalar

1 Introduction

At the end of the current decade, processor chips will probably contain 50 millions of transistors. One way to make use of these resources is to integrate a multiprocessor in the chip [12] [8]. A straightforward implementation would be to duplicate 2, 4 or 8 times an actual processor. This would lead to unnecessary copies of rarely used operators like the floating point ones (not all the programs use floating point data) or integer dividers, even though these operators are nowaday a small part of the total transistors. In the case of the memory hierarchy, having one separate cache per processor would give each a small share with a higher miss rate than with a single shared cache. Clearly, an on-chip multiprocessor, to be scalable, must make use of some shared and some duplicated resources. The main architectural issue is to decide what should be shared and organize the sharing so that each processor throughput is maximized. The main advantage in sharing is that it leaves more place on-chip for the memory hierarchy. This favours the IPC. But sharing causes selection problems all along the pipelines as we have experienced in previous designs [6] [7]: selecting the fetching thread, the issued instructions, the operators sources, the written results, the loads and stores. This either degrades the cycle or lengthens the pipelines which degrades the IPC.

Tullsen et al. [20] have proposed and measured the efficiency of an implementation of their previously published SMT architecture [19]. An SMT (Simultaneous MultiThreading) architecture can be viewed as a single processor architecture in which some features are duplicated (at least the program counter) to allow true (not time-shared) multiprocessing. In the former multithreaded machines like HEP [13], MASA [10], APRIL [1] and nowadays in the Tera machine [2], the whole processor except the program counters is shared by the threads (the register file is either a shared pool or, in APRIL, composed of separate banks). The fetch, issue and execution pipe stages logics each concern a single thread at a time. In [19], it is proposed to share some resources (e.g. functional units) and to duplicate others (e.g. fetch path). The fetch logic serves all the threads in parallel. The issue logic selects instructions from multiple threads and schedules them to available shared functional units. The execution logic runs instructions from multiple threads. In [20] implementation, the fetch logic comes back to shared.

According to [20] results, the design achieves a "2.5

fold improvement over" a single threaded "superscalar processor with similar hardware resources". What the simulation shows is that the IPC term of the processor performance equation is such improved. On another hand, the authors state that their implementation, which first aims to be as close as possible to standard superscalar processors, allows an unmodified cycle term. To achieve this, they have organized fetch to serve each thread in turn and merged the decoded and renamed instructions in two queues. Fetching, decoding, renaming, issuing, computing and terminating are performed as in a single thread processor. The most part of their SMT architecture is shared. Their implementation is closer to the previous multithreaded architectures we have just mentionned than the first SMT architecture proposal. In a certain way, in their implementation, multithreading is used to provide more ILP to a standard superscalar architecture. Only two main features are added to the traditional design for out-of-order and speculative execution [11]. The first one is the fetching thread selection and the second one is an enlarged register file which includes each thread register set plus a common pool of renaming registers. The register file was assumed by the authors to have a two cycles access time, implying two new stages in the pipeline.

In this paper, we propose to investigate another style of SMT implementation which is closer to an on-chip multiprocessor. Our proposal is just in between a single processor extended to multiprocessing facilities and a multiprocessor restricted to share some parts of the hardware. As shown in [17], a multiprocessor performs better than an APRIL-like [1] multithreaded processor (in APRIL, a single thread is run at a time; when it experiences a cache miss, a thread switch occurs). In [12], it is shown that a multiprocessor made of low issue processors performs better than a high issue superscalar single processor. Of course then, it is clear that a multiprocessor made of high issue superscalar processors would perform even better. Is it worth it then to investigate multithreaded architectures further more? The answer is probably no for non-SMT (APRIL-like) architectures. What about SMT architectures? Do they have a chance to be competitive with a multiprocessor? We believe the answer is that the closer you get from the multiprocessor, the better the performance. Why then not simply relying on full duplication (except for the on-chip memory hierarchy after the first cache level)? It is well known that, among the functional units in a superscalar processor, some are under-utilized, such as the dividers or the floating point operators when

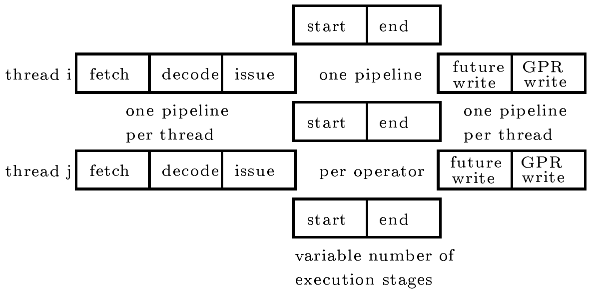

Figure 1: **Tipi's pipelines**

integer code is run. Operators sharing is a nice opportunity to exactly adjust the number of each unit copy required by the number of on-chip processors. It enhances multiprocessor scalability. Another argument in favour of sharing given in [19] is that functional units sharing allows a dynamic partitioning of these units among threads, whereas in a multiprocessor, the partitioning is static. In other words, adding an operator to a shared organization adds the operator to potentially all the threads. Adding an operator to one of the processors in a multiprocessor has no effect on the others.

But such a sharing raises many problems due to the communication between the non-shared parts and the shared ones. The restricted multiprocessor or the extended single processor should be designed in such a way that operator sharing does not increase neither cycle width nor pipeline length.

Our design does increase pipeline length (one more stage is added to the main pipeline), so, to compensate, it reduces cycle width, taking advantage of the operators sharing. The very short cycle gives a two cycles latency to a simple adder operator. In a single processor, such a pipelining would be useless because the instruction flow would, in most cases, either issue adjacent instructions not using the same operator or dependent on each other. In a multiprocessor, such pipelining is an equivalent of the operator duplication. To achieve a so short cycle, instead of keeping the pipeline main structure unchanged in respect with actual single thread processors, we have adapted the Tomasulo's algorithm [18] to very short stages pipelines (even shorter than in the famous DEC 21164 [3] [9]). In this respect, all the features requiring some dynamic allocation have been replaced by more simple structures. Our design should achieve a good performance because it has a very short cycle (10 levels of logic gates per stage, not including the inter-

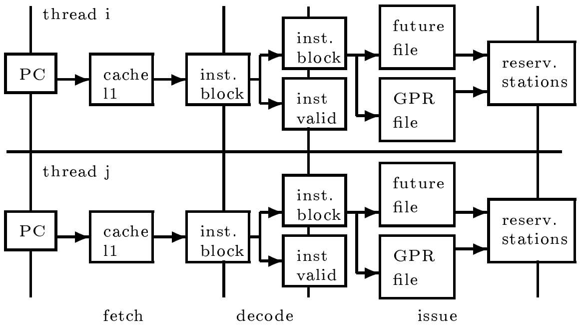

Figure 2: **fetch, decode and issue stages**

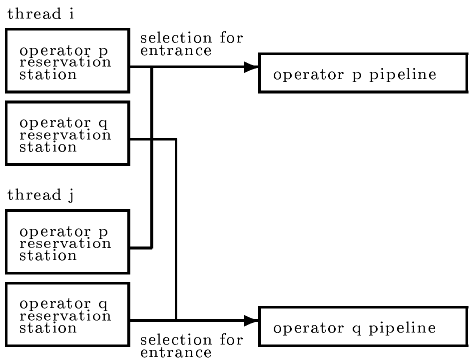

Figure 3: **operator pipeline entrance logic**

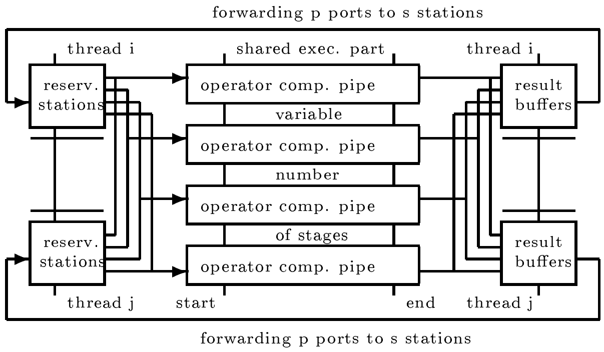

Figure 4: **start, execute and end stages**

stage latching logic), a middle size pipeline (7 stages for simple instructions), a high superscalar degree (a full instruction cache block per cycle is the peak issue rate) and a good thread (i.e. multiprocessor) scalability (8 threads would fit in the short cycle).

2 An overview of the Tipi processor

Figure 1 is a general view of Tipi's 7 stages pipeline, showing the separation between the duplicated part (one pipeline per thread) and the shared one (one pipeline per operator, shared by all the threads).

To provide a high enough fetch bandwidth [16], separate per-thread first level instruction and data caches are used. Fetching, decoding and issuing are handled in parallel for all the threads (see figure 2). Simultaneous out-of-order issue has not been considered. In-order issue gives less throughput but it may be implemented with the very short cycle requirement and the resulting pipeline has 3 stages only from fetch to issue (no rename stage).

After issue, instructions must be driven from the per thread stations in which they were issued to shared operators pipelines [7]. Logic can be considerably simplified if the set of operators is considered as the set of registers, being addressable as in the MIPS R10000 [21]. In this processor, the original opcode derives a 4-bits functional unit number which is placed in the cache as an extension. We propose to replace this opcode derivation with an operator allocation. Such an allocation could be static (the operator number is part of the opcode) or dynamic (the operator number is allocated -not derived- each time the instruction opcode is written in the second level instruction cache). If such an allocation is implemented, per thread issue places instructions in their single allocated station. Operator pipeline entrance logic is a simple selection among all the threads single stations (see figure 3). The allocation algorithm uses a virtual numbering of the architecture operators and the implementation maps it to the real set, with a per thread mapping. This mapping allows different implementations with full or partial operator set. It also reduces the conflicts between threads sharing duplicates of an operator (e.g., the algorithm allocates copy number 0 to every thread and the mapping gives each thread a private copy).

Operation start, execution and end (see figure 4) take a longer time than in a single thread pipeline. Station selection and per thread results ordering are logics added to the basic operator computation. They add a new stage to the pipeline (an operation is started in the first half of the first stage, computed from the second half of the first stage until the first half of the last stage and the result is driven to its thread bus in the second half of the last stage).

Let us now describe each pipe stage in more details to show that the cycle can really be very short.

3 Fetching

The main tradeoff in fetching logic concerns the instruction cache. In [20], a shared cache is assumed, with a single read port. The size of the cache must be fixed according to the number of threads even though code sharing may occur. This leads to an increased access time and bandwidth problems. Because we favour the cycle, we choose small private caches. The advantages in respect with the shared cache are that the fetch bandwidth is larger (more ports may be added to the shared cache but this increases access time) and the access time is shorter. The drawback is the higher miss rate. In order not to penalize the CPI too much, the miss penalty must be kept small with an on-chip second level cache. This one can be shared, associative and large. This on-chip two levels memory hierarchy is comparable to the DEC 21164 one, adapted to multithreading for the second level.

A second advantage of separate caches is that they can be direct-mapped. A shared cache has to be set associative, which requires a tag comparison to choose a block in the set, enlarging the access time. A final advantage of small first level caches over large ones is that they may be addressed with a smaller part of PC. This allows both the usage of the virtual address and a faster preparation of the next PC value. In our design, next PC is computed in two cycles, with a carry select method. While cache read is performed, the next lower part is prepared in parallel with upper part slices assuming carry and no-carry. In the next cycle, the upper part is formed, soon enough for tag check. Figure 5 shows the pipelined PC computation and instruction cache access and tag check.

Fetch stage critical path should come from instruction cache access time. It includes address decoding, memory cell read and data driving. Decoding and reading costs from 3 to 5 gates levels according to the cache size (between 128 and 1024 blocks). Driving delay cannot be estimated with simple gates levels. We expect the total delay to be at most 10 gates levels. In comparison to traditional caches designs, our choices reduce address decoder size (small cache), avoids word selection and places tag check in a different stage. If timing problems still remain, cache data can be splitted, separating opcode constant field. Then, reading other fields than the constant value can extend beyond the first stage. Only conditional branch target computation, which is based on the constant field, is critical as will be seen in the next section.

Next PC computation critical path in the fetch stage does not come from its incrementation. The target address computation, implied by a conditional PC-relative branch, is more critical since it requires an adder, at least for the next PC low part. Its complexity is analyzed with the decode stage in next section.

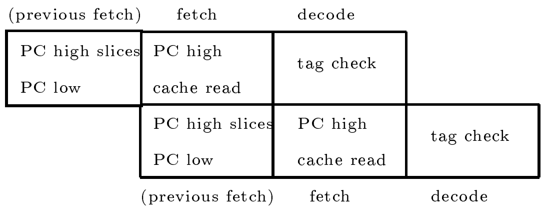

Figure 5: **PC sliced computation**

Control flow prediction is started in the fetch stage. This prediction is not based on a cache of targets addresses. Such a cache is not adapted to our short cycle design because, if it can be read in parallel with fetch, its tag check has to be postponed to the decode phase, as for the instruction cache, giving a too late prediction. Instead, we place a cache of branch directions in the fetch stage [14]. Such a cache is much smaller than a target address cache, which is important because it must be duplicated for each thread. The cache is accessed in parallel with the instruction cache and delivers a branch direction prediction for each word of the fetched block. The read predictions must be ready when the decode stage starts, in order to leave enough time for the predicted PC to be computed. In other words, the target cache read may not overlap on the decode phase. It is less a problem than for the instruction cache because the read block contains two bits words which requires less driving time.

Two more treatments are performed during fetch stage: invalidating the fetched block part that is on the PC pointer left side (the complexity is proportional to $\log_2 n$, with n being the number of instructions in a cache block); tagging unconditional and validated branches and tagging conditional, predicted taken and validated branches. This last logic is the critical one because it follows the invalidation and because it uses control flow instructions pre-decoded bits, read from a PC addressed cache. The critical path is made of the pre-decoded bits read followed by an intersection with the prediction bits and the validation ones. It represents from 5 to 7 gates levels when $n = 8$, according to the cache size. We expect the driving requirements to fit in the remaining time fixed by the 10 gates levels limit.

4 Decoding

The decoding phase must select the instructions to be issued. The main problem is to find the leftmost predicted taken branch in the fetched block and to invalidate the following instructions (see figure 6b). Using cached pre-decoded bits can significantly reduce the logic. The selection of the branch is, for $n = 8$, an 8-to-1 multiplexing (3 gates levels). The part of the fetched block placed after the branch must be invalidated. Of course, invalidation and selection can be performed in parallel. Invalidation is a $\log_2 n$ intersection based logic (3 gates levels for $n = 8$). Intersecting fetched block validated prefix and suffix gives the set of instructions to issue (4 gates levels).

Predicted next PC is computed after the leftmost branch detection. This choice avoids the need of n adders to compute the n possible target addresses. A return address is predicted with a stack. This part of the prediction should not be critical. Indirect branches that are not return instructions are predicted with a hint given by the opcode as in the Dec Alpha ISA [4]. A conditional branch requires a PC-relative target address computation. It is performed in two cycles with a carry select method. In the first cycle, byte wide adders are used to prepare both the low part of PC and the upper part slices presuming carry and no carry. An 8 bits carry select adder can be built with only 4 gates levels. Thus, an up to 16 bits next PC low part is computed in 9 gates levels (next PC is a selection among incremented PC, predicted target address and corrected target address; the most critical one is the predicted target address because it contains an addition; its critical path has 3 gates levels for leftmost branch selection, followed by 4 gates levels for bytes additions and ended by 1 gate level for a final upper byte selection; the last selection for next PC adds one more gate level).

When a control flow is predicted (during a decode phase) for a thread, its on going fetch is cancelled (see figure 6a). Thanks to the decode stage that includes the full prediction logic, fetch-to-prediction latency is only one cycle long. Nevertheless, this reduces significantly each thread fetch bandwidth. Enlarging the cache blocks increases the bandwidth on one side and decreases it on another side. Future simulation should point out which block size is the best.

The decode stage also contains the instruction cache tag check logic. Because it would be too long to translate the virtual address, the instruction cache keeps virtual tags. To prevent from synonyms, the cache is direct mapped (this also shortens access time and allows keeping tag check out of the critical path).

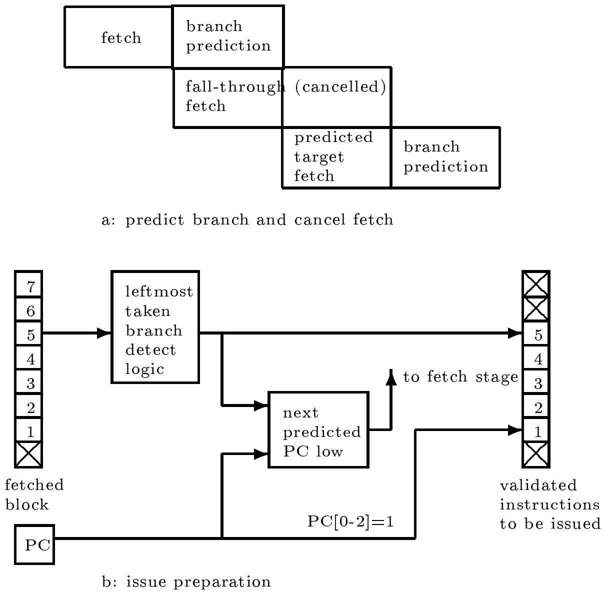

Figure 6: **decode stage**

A cache miss stops the thread fetches. The second level cache is accessed after one or more arbitration cycles (here again, a future simulation should help to fix a maximum number of threads after which conflicts are too frequent). In parallel, address translation takes place, after which set selection occurs.

The first level tag comparison is performed in 7 gates levels (binary tree equality comparator) for an up to 64 bits tag. Intersecting with the block valid bit gives a miss status in 8 gates levels.

The block, once decoded, is appended to the issue queue. Fetch and decode stages are frozen when the queue is full.

5 Issuing

Validated instructions in the issue queue head block are issued when their resources are available. The issue decision depends on the following conditions:

1. the instruction entry in the queue block is marked valid;
2. its allocated station is empty;
3. its destination register is not under computation (future file copy is not in use);
4. the reorder table has a free block;
5. preceding instructions in the same queue block are either invalid or issued (in-order issue);

6. preceding instructions in the same queue block do not use the same station;

7. preceding instructions in the same queue block do not use the same destination register.

The four first conditions may be checked in parallel in constant time (two gates levels for intersection). The three last ones are also checked in parallel, but in a time proportional to $\log_2 n$. The station usage check employs $n*(n-1)/2$ station number comparators (e.g. 5 bits wide for 32 operators) followed by an intersection of the comparisons. It is computed in 7 gates levels for $n = 8$. The destination register usage check is done exactly the same way. The final issue condition is a two gates levels intersection, giving a total delay of 9 gates levels.

In parallel with issue decision, operands are read from the future file ($2n$ read ports). A future file [15] is not as efficient as a pool of renaming registers because a single copy is available for each register. However, most of the name dependencies can be removed with a single copy and we expect that the remaining ones do not reduce single thread IPC significantly enough to overcome the simplicity advantage which is payed back in the cycle width. At the end of the stage, the forwarded results are matched with the issued instructions sources. The file read logic contains a source register number decoding and a register cell read, for a total of 4 gates levels. Driving the operands to the stations requires a selection between constants and register sources and a switching to the addressed station (each station selects its sources from the $2n$ ports in a time proportional to $\log_2 n$). Eventually, forwarding adds one more selection for a total of 9 gates levels when $n = 8$.

Still for the same reason of simplicity to keep the cycle short, the reorder buffer (not shared; one per thread; see figure 7) is organized as a FIFO of blocks. A first drawback is that the buffer is larger than necessary because it contains entries for invalidated (non issued) instructions. Another inconvenient is that instructions retirement may not cross a block boundary (only the tail block is examined for retirement). The first advantage is that the buffer allocation and liberation are simpler (0 or 1 block per cycle). The second one is that the retirement logic is also simpler. The retiring decision is proportional to $\log_2 n$.

6 Computation start

Each thread has one reservation station per operator. For each operator, its threads stations compete for pipeline entrance. The selection is priority

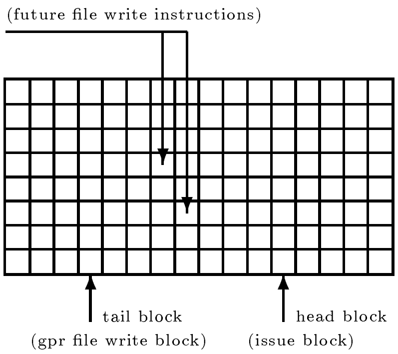

Figure 7: **the reorder buffer**

based among the ready instructions. It is proportional to $\log_2 t$, where t is the number of threads. For $t = 8$, 4 gates levels are necessary to drive the selected operands to the operator input (1 level for the first multiplexer rank control and 3 levels for multiplexing).

The operation itself is distributed on at least two stages, to keep the cycle very short. For example (see figure 8), an addition starts with byte wide adders (4 gates levels) in the second half of the starting cycle and ends with a carry based selection in the first half of the next cycle (3 gates levels for a 64 bits addition). This leads to a one cycle delay between dependent instructions. The implied CPI reduction in a single thread processor would nullify the cycle width benefit of such pipelining, but in a multithreaded environment, threads interleaving helps filling the high rated pipeline. Throughput relies on the number of available threads more than on the number of single thread dependencies.

7 Computation end

For each operator, a set of t buffers is provided for the result. The value is driven to the correct buffer according to the thread number, during the second half of the computation ending cycle (see figures 4 and 9). The driving operation is proportional to $\log_2 t$ (3 gates levels when $t = 8$).

For conditional branches, the prediction is checked and the corrected PC is formed and buffered if necessary (the comparison logic, i.e. 7 gates levels for 64 bits addresses, is distributed on two cycles, as for the adder).

The results are forwarded to the stations either in the computation end stage or during the future file write cycle. The latter case advantage is that it requires a limited number of comparators ($s*p$ for each

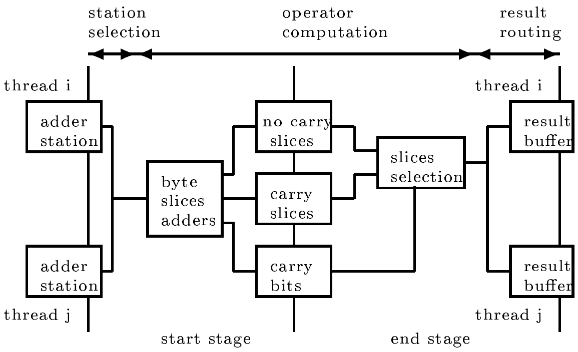

Figure 8: **a sliced adder**

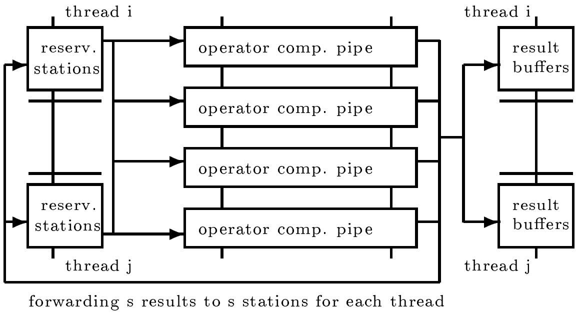

Figure 9: **forwarding**

thread, where s is the number of stations and p is the number of future file write ports). Its main drawback is that it increases latency (two cycles to separate dependent instructions). In the former case, s^2 comparators per thread are needed. In the first half of the computation end stage, the destination is compared to the expected sources (3 gates levels). In the second half, the ready results are forwarded to the stations (proportional to $\log_2 s$; remember that there is one station per operator, hence s results; with 32 operators this leads to 5 gates levels for result selection and a total of 9 gates levels to fix station operands inputs, as was figured in the issue stage description). We may think of two implementations: one which is devoted to many threads with a long latency (see figure 4), and another limiting threads to 4 (see figure 9). A future simulation should show if enough throughput can be reached in the many threads implementation.

8 Writing in the future file

After computation, results are placed in per thread buffers. A selection must be performed if less write ports than buffers are available. This selection can be simplified if each port is linked to a fixed subset of the buffers. Operator allocation and virtual to real operators mapping can be used to reduce ports conflicts. Selection is then proportional to $\log_2 (s/n)$ and forwarding, if placed in this stage, is proportional to $\log_2 n$. With 32 operators and 8 ports ($s = 32$ and $p = 8$), placing each result on its port bus takes two gates levels. Forwarding comparison takes 3 more gates levels and it is followed by a 3 gates levels selection, for a total of 8 gates levels to present the forwarded value to the station input final choice.

For each selected result, its thread reorder buffer is updated. A corrected PC value (result of a mispredicted control flow instruction) is copied to its instruction entry in the reorder buffer.

9 Writing in the GPR file

The tail block in the reorder buffer is examined to select the writing instructions. An instruction result is written in the GPR if none of the preceding instructions in the same block has raised an exception. This check is proportional to $\log_2 n$ (it is very similar to the issue conditions logic). The destination field in each entry of the tail block serves to address the future file (read ports) and the GPR file (write ports). The total copy delay is 9 gates levels. In case of an exception, instructions placed after it in the tail block are invalidated and only the instructions placed before the exception write in the GPR file. The thread resources are reset. The future file is emptied (or reset with the GPR file content), as are the stations and the reorder buffer. The exception address, taken from the reorder buffer, is sent to the thread PC update logic.

10 Loads and stores

Data caches are organized as the instruction caches. A first level is made of small private caches and a second level is shared.

Loads and stores are treated quite differently than other instructions. First, because the first level caches are private, loads and stores travel through per thread pipelines. Second, because loads and stores hazard control relies on the memory address which needs computation, issue decision is splitted in two stages (see figure 10).

A first issue decision is taken as for other instructions, according to the same conditions stated in section 5 (in-order issue). The issued loads and stores compute their address low part in their first execute stage (same complexity as an adder operator first cycle). Then, the second issue decision takes place (in-order issue) in parallel with cache access. Stores lock

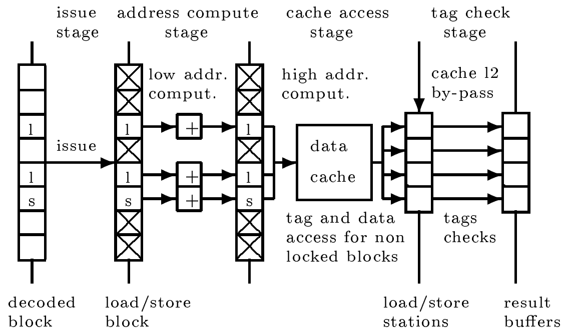

Figure 10: **loads and stores**

the block they address in the cache and a load or a store may not be further executed if the block it addresses is locked or being locked. In such a way, we proceed with the cache as with the register file, the accessed block playing the role of the destination register. The issue condition has the same $\log_2 n$ complexity as the destination register usage check presented in the description of the issue stage. It requires 8 to 10 bits equality comparators (instead of 5 bits ones) according to the number of cache blocks varying from 256 to 1024.

The tag and data read from the cache for issued loads and stores are buffered after a by-pass selection coming from cache level two, based on thread number and cache block number. This by-pass ends a previous cache miss.

If not enough cache ports are available, a selection is necessary. It takes place before cache access. For $n = 8$ and 4 cache ports, one gate level is required to select which loads and stores in the issuable block are issued. For loads, a word selection is performed. The total read delay is 8 gates levels, including the second level cache by-pass selection. Here, as in the instruction cache, data driving should add a lot to the delay. In such a case, data read operation may overlap, with word selection, on the next cycle (however, this increases load latency). Tag access, which does not require any selection, can be performed within the cycle, without overlapping.

The tag check stage loops until the correct tag is recognized. The corresponding station pipeline remains frozen until hit. The correct tag either comes directly from cache level one (hit) or is by-passed by cache level two (miss).

Loads and stores end their execution as other instructions. Loads write the loaded data in the future file and after reordering, in the GPR file. Stores write the data in a store FIFO, written in the cache after reordering.

11 Conclusion

In this paper, we have presented an SMT implementation closer to a multiprocessor than in [20]. Our design has the following advantages:

1. its cycle should be very short (the necessary logic is limited to roughly 10 gates levels per stage, for 8 threads and a superscalar degree of 8 per thread);

2. it should scale well with the number of threads;

3. it should scale well with the number of operators;

Its main innovations, essential to the cycle reduction, are the statically or dynamically allocated operators, with virtual numbering and hardware per thread real mapping; the adaptation of the Tomasulo's algorithm to multithreading and superpipelining; the sliced PC and addition computations, which help keeping the cycle very short despite the multithreading routing overhead; the load/store instructions handling with hazards solved as for the registers and with misses using a by-pass technique.

A possible drawback would be an unexpectedly low per thread IPC. A future simulation is essential to show that it keeps high enough.

References

[1] A. Agarwal, B.H. Lim, D. Kranz and J. Kubiatowicz: APRIL: a processor architecture for multiprocessing. 17th ISCA, 1990, pp104-114.

[2] R. Alverson, D. Callahan, D. Cummings, B. Koblenz, A. Porterfield and B. Smith: The Tera computer system. International Conference on Supercomputing, 1990, pp1-6.

[3] B.J. Benschneider et al.: A 300-mhz 64-b quad-issue CMOS RISC microprocessor. IEEE J. of SSC, vol. 30-11, nov. 1995, pp1203-1211.

[4] Alpha Architecture Handbook. Digital, 1992.

[5] J.H. Edmondson, P. Rubinfield, R. Preston and V. Rajagopalan: Superscalar Instruction Execution in the 21164 Alpha Microprocessor. IEEE Micro, april 1995, pp33-43.

[6] B. Goossens and M. Akil: MT: a multithreaded 64 bits RISC CPU. 2nd Parallel Computing Technologies Conference, Russia, 1993, pp47-62.

[7] B. Goossens and D.T. Vu: Multithreading to improve cycle width and CPI in superpipelined superscalar processors. 2nd IEEE ISPAN symposium, 1996, pp36-42.

[8] B. Goossens and D.T. Vu: On-chip multiprocessing. 2nd EuroPar, 1996, LNCS 1124, pp789-796.

[9] P.E. Gronowski et al.: A 433-mhz 64-b quad-issue CMOS RISC microprocessor. IEEE J. of SSC, vol. 31-11, nov. 1996, pp1687-1695.

[10] R.H. Halstead and T. Fujita: A multithreaded processor architecture for parallel symbolic computing. 15th ISCA, 1988, pp443-451.

[11] J.L. Hennessy and D.A. Patterson: computer architecture: a quantitative approach. 2nd edition, Morgan Kaufmann, 1996.

[12] K. Olukotun et al.: The case for a single-chip multiprocessor. ASPLOS 7, 1996, pp2-11.

[13] B.J. Smith: Architecture and applications of the HEP multiprocessor computer system. SPIE Real time signal processing IV, 1981, pp241-248.

[14] J. Smith: A study of branch prediction strategies. 8th ISCA, 1981, pp135-148.

[15] J.E. Smith and A.R. Pleszkun: Implementing precise interrupts in pipelined processors. IEEE Trans. on Computers, vol. 37-5, 1988, pp562-573.

[16] M.D. Smith, M. Johnson and M.A. Horowitz: Limits on multiple instruction issue. ASPLOS 3, 1989, pp290-302.

[17] R. Thekkath and S.J. Eggers: The effectiveness of multiple hardware contexts. ASPLOS 6, 1994, pp328-337.

[18] R.M. Tomasulo: An efficient algorithm for exploiting multiple arithmetic units. IBM J. of Res. and Dev., vol. 11-1, 1967, pp25-33.

[19] D.M. Tullsen, S.J. Eggers and H.M. Levy: Simultaneous multithreading: maximizing on-chip parallelism. 22nd ISCA, 1995, pp392-403.

[20] D.M. Tullsen et al.: Exploiting choice: Instruction fetch and issue on an implementable simultaneous multithreading processor. 23rd ISCA, 1996, pp191-202.

[21] K.C. Yeager: the MIPS R10000 superscalar microprocessor. IEEE micro, april 1996, pp28-40.