Typing the ISA to Cluster the Processor

Bernard Goossens

LIAFA, Université Paris 7, 2 place Jussieu, Paris Cedex 05, France

Abstract

We propose a separation of the processor units to decrease the communication paths lengths. The motivation comes from the increasing importance of intrachip signalling delays. The technique consists in separating the instruction set into types and the die into corresponding units. Every type is able to fully handle data and pointer computations as well as typed address pointers. Hence the processor is composed of very independent machines requiring no inter-machine communications. We show that the fetch path length can be divided by two and the data load path length can be decreased of 1/3.

Key words: Microarchitecture, Clustered Processor, Typed ISA, Fetch Path.

1 Introduction

Todays processors are built around 5 units: one instruction memory unit including an instruction cache, a fetch mechanism and a prediction subunit; three computing units (integer, floating point, graphic: the graphic unit can be further subdivided into an integer vector unit and a floating point vector unit); one data memory unit including load and store buffers and a data cache. Figure 1 left part shows how such a processor is organized on a die. The die is divided into three equal area parts, one devoted to the instruction memory unit, another devoted to the data memory unit and a third one containing the core, i.e. the three computing units. The size of the instruction cache has a direct impact on the processor cycle, with today a pipelined fetch unit delivering one fetch per cycle with a two cycles fetch latency [8,6].

Table 1 gives a *cacti* [15] estimation of the cache access times for different cache sizes and chips technologies (the times given are the data side ones in nano seconds, without output driver; computations parameters are: 128 output bits, 32 address bits, blocks of 32 bytes for caches up to 128KB and

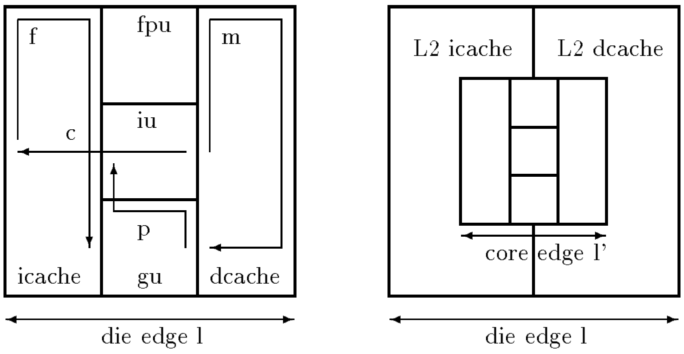

Fig. 1. **Two processor dies**

64 bytes for larger caches, direct mapping for caches up to 8KB, associativity 2 for 16KB caches, 4 for 64KB caches and 8 for larger caches).

If we assume a clock cycle doubling for each technology step, we can see that in 1998, a full fetch including cache block read and instruction selection and slotting, in a typical 16KB cache was possible in a cycle (as in the DEC 21164 [3]). In 2000, a fetch in a typical 64KB cache in a 600Mhz cpu (as in the DEC 21264) had a two cycles latency with a cache block read time of one cycle, hence the necessity of pipelining the fetch path (if the cache read time is larger than a cycle, wave pipelining [5] can be used to provide a cache throughput of one read per cycle as long as accurate addresses can be provided at the same rate). In a 1Ghz 0.18μ cpu, the first level cache must be reduced to 8KB to allow a single cycle read (as in the Pentium III [11]; the Pentium 4 [12] and the AMD Athlon [1] have larger L1 caches accessed in 2 cycles).

Moreover, driving the fetched instructions to the computing units is also a concern in todays cpus. For example the Pentium 4 dedicates a full cycle for this purpose. Tomorrow, in a 3Ghz cpu including an 8KB L1 instruction cache, if the path connecting the fetch machine to the computing units is not shortened, the drive time on its own will take two full cycles (to be added to the two cache read cycles).

This paper gives two hints to reduce fetch and load/store read path lengths in order to cope with the increased drive time. This paper is linked to the various works concerning processor clustering [4,10]. The main difference between the present approach and previous proposals is that we modify the ISA (Instruction Set Architecture). The consequence is that clustering is not only performed on the set of functional units but also on the fetch path and on the load/store path.

The paper is organized as follows: next section presents a two levels cache organization allowing simultaneous accesses to L1 and L2. Section 3 describes

Table 1
Caches data path times

techno.	1KB	4KB	8KB	16KB	64KB	256KB	1MB
0.35μ	0.97	1.15	1.27	2.03	2.99	4.46	7.93
0.25μ	0.70	0.82	0.91	1.45	2.14	3.18	5.67
0.18μ	0.50	0.59	0.65	1.04	1.54	2.29	4.08
0.15μ	0.42	0.49	0.54	0.87	1.28	1.91	3.40
0.13μ	0.36	0.43	0.47	0.75	1.11	1.66	2.95
0.1μ	0.28	0.33	0.36	0.58	0.86	1.27	2.27

a new die layout clustered by data types. Section 4 explains which are the necessary ISA modifications. Section 5 discusses the paper proposal and compares it to the known works on cpu clustering, multithreading and on-chip multiprocessors.

2 A Two Levels Simultaneously Accessed Cache

On the die example presented on figure 1 left part, the fetch path is labeled 'f'. The fetch critical path goes from the center of the fetch unit (labeled *icache* on the figure) left edge where the pc is assumed to be located. It travels (say) to the fetch unit upper right corner where the fetched block is assumed to reside and the fetched graphical instructions are propagated to the graphic unit (labeled *gu*) entrance which is assumed to be at the center of the unit left edge. This $5l/3$ path, where l is the die edge length ($l/2+l/3+2l/3+l/6$), corresponds to a worst case, a best one having an $l/3$ length (integer instruction placed at the center of the cache area). In a cpu today, this work is performed in two cycles, the first one being devoted to cache read (length $4l/3$) and the second one to instruction slotting (length $l/3$ and some decoding and selection logic). In such a cpu, the cycle is fixed to leave enough time for cache read, i.e. for a signal to cross a length $4l/3$ distance. Because the die edge l should be increased ([2]; 20mm in 2000, 28mm in 2005 and 40mm in 2010), as long as the cache read path is linked to l, the cycle time, if equal to the cache read time, should increase too (in other words, if we use the edge length increase to fill the die with cache, we increase the cycle time).

In the case of a memory access, an inter-computing-units communication takes place. The address computation is performed in the integer unit while the access itself may concern another unit register. The load path is labeled 'm' on the figure. It is longer than the fetch path because it involves a round trip. On the die example, the worst load distance is $2l$ starting at the center of the integer unit right edge (load address), going up to the data cache upper right corner where the block to be loaded is assumed to reside and going back to the center of the loading computing unit left edge, which is assumed to be the graphic one in the worst case ($l/2 + l/3 + 2l/3 + l/6 + l/3$). With today signalling delays, loading has at least a two cycles latency.

A first attempt to shorten the fetch and the load/store paths is to separate each cache in two levels. The die can be organized as shown on figure 1 right part. It includes a core corresponding to the left die, surrounded by two level 2 caches. The die edge is l (scaled to the technology process) and the core edge is l', with a wire delay of 1 cycle for a $4l'/3$ length (we link the cycle time to the l' length which should decrease as technology is refined). Such a design is close to the DEC 21164 or the Pentium III (except that the L2 cache in these processors is a unified cache). This two levels cache organization with a small L1 cache was abandoned by DEC because the L1 cache had a too high miss rate and the first level miss detection incurred a too high penalty for the second level access time. It is still in use in the Pentium III but is also abandoned both in the AMD Athlon and in the Pentium 4 with a 64KB cache for the former and a 12Kops trace cache for the latter (the trace cache contains fixed length micro operations; it has a capacity roughly equivalent to a 64KB cache).

But as table 1 shows, a 64KB cache capacity in a 2Ghz CPU has a three cycles read latency and five cycles in a 3Ghz CPU. On the other end, a cost effective two levels hierarchy starting with a small capacity L1 cache requires a reduced L2 access time. This can be done by performing both accesses (L1 and L2) simultaneously (the L2 access is stopped if the L1 hits) which implies keeping the L2 instruction cache distinct from the L2 data cache to avoid contention. In such a design, the L2 instruction cache can preferably be a victim cache as in the AMD Athlon (in case of a miss, the L1 cache loads the missing line and transfers to L2 the replaced line). In such a way, L1 miss detection does not add any penalty to L2 access time. In this design, a fetch that hits in the first level crosses a length $5l'/3$ with $4l'/3$ in a single cycle. For a L2 fetch, the distance is $2l - l'/3$ ($2l - l'$ for L2 access and $2l'/3$ to propagate L2 block to (say) the fpu). If we assume $l = 2l'$ as drawn on the figure, the L2 read ($3l'$) can be performed in 2 cycles (one more cycle is needed to slot instructions in their units, as in the case of a first level hit). As l' is decreased and l is increased, L2 reads may last much more than two cycles in finer technologies. When the gap becomes too important, it may be necessary to separate L2 in two levels, having three levels on-chip. However, by keeping the L1 caches constant in capacity, the l' length can scale with the technology allowing the cycle to scale too.

This is true only if the core does not increase (l' is not increased for architectural reasons). Unfortunately, actual trends with higher superscalar degrees is to add more and more functional units, along with new specialized ones such as the SIMD operators and bigger branch prediction hardware. Hence l' should not remain constant, implying that signal propagation inside the core itself may become a problem.

3 A New Die Layout

Figure 2 shows an alternative die organization that has three main differences with the layouts on figure 1: the caches first level have been duplicated, each copy being local to a computing unit; the fetch subunit and the prediction subunit have been duplicated with each copy placed on the left edge of each L1 instruction cache (there are three pc); eventually, the branch unit has been duplicated in every computing unit (the duplications are not apparent on the figure except for L1 caches).

The 'f' path starts from the center of an L1 unit left edge, goes up to the same L1 unit upper right corner and ends at the center of the corresponding computing unit left edge. The length is $2l'/3$, due to the locality of the fetch ($l'/6+l'/3+l'/6$). This is less than 1/2 of the figure 1 right die 'f' path length. In case of an L1 miss, the 'f' path length starts from the center of the L1 (say) graphic unit (gu) left edge, travels up to the L2 upper left corner (say), comes back to the starting point and ends at the center of the graphic computing unit left edge. Hence, it has length $2l$ ($2*(l'/3+l/2+(l-l')/2)+l'/3$) which is $l'/3$ longer than figure 1 right die 'f' path (the difference comes from the starting point of the fetch which is equidistant from each corner of the right die on figure 1 but not on figure 2).

The 'm' path starts from the center of any computing unit right edge, goes to the corresponding L1 data unit upper right corner and travels back to the center of the same computing unit left edge ($l'/3 + l'/6 + 2l'/3 + l'/6$). This is a length $4l'/3$ path (1/3 less than the right die 'm' path which has length $2l'$). In case of an L1 miss, the L2 access path has length $2l + 2l'/3$ ($2*(l'/3 + (l - l')/2 + l'/3 + l/2) + l'/3$). This is $2l'/3$ longer than right die 'm' path. The reason is that the address computation is local to the loading unit in the die on figure 2 instead of being performed in the integer unit in the right die on figure 1.

To fairly compare the two layouts, we must take in account that the L1 caches are partitioned in the new layout. This means that each unit has a private part that is 1/3 of the total L1 cache size. The cache miss rate is affected by this capacity reduction. Let r_1 be the L1 miss rate on figure 1 right die and r_2 the L1 miss rate on figure 2 die. The average fetch length on figure 1 right die is $5l'/3*(1-r_1)+(2l-l'/3)*r_1$ and the average fetch length on figure

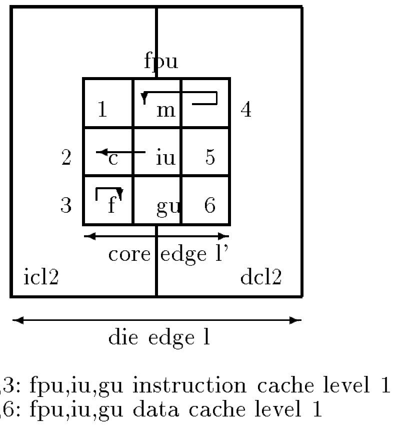

Fig. 2. **A new die layout**

2 die is $2l'/3 * (1 - r_2) + 2l * r_2$. The latter is better than the former as soon as $l < (3 - 6r_1 + 2r_2)/(6(r_2 - r_1))l'$. If for example $r_1 = 0.08$ and $r_2 = 0.16$, the new layout is better than the old one if $l < 5.9l'$ (when l gets very large compared to l', the L2 time becomes dominant; in such a case it is necessary to insert an intermediate level between L1 and L2; as L1 on figure 2, this intermediate level can be clustered).

The two layouts must also be compared for inter-unit communications. An inter-unit communication is involved when a conditional branch or indirect jump has not been correctly predicted. In this case, the corrected continuing fetch address is transfered from the branch unit to the prediction subunit (the branch unit can either be a separate unit or as assumed on figure 1, part of the integer unit). The prediction correction critical path, labeled 'c' on figure 1, can be assumed to be of length $2l/3$ ($2l'/3$ on the right die), going from the center of the integer unit right edge to the prediction subunit, located nearby the pc register, i.e. at the center of the fetch unit left edge. On figure 2, the 'c' path runs from the center of any computing unit right edge to the center of the corresponding L1 unit left edge. The path has a length of $l'/3$ which is half of the path on the figure 1 right die.

Also concerning the branch unit, the new set of predicated instructions [7] can lead to predicate interunit communications (e.g. a predicate obtained from the comparison of two floating point numbers and used in the branch unit). The predicate propagation critical path, labeled 'p' on figure 1, can be assumed to have length $2l/3$ ($2l'/3$ on the right die), starting from the center of the (say) graphic unit right edge and ending at the center of the integer unit left edge ($l/6 + l/3 + l/6$). Because the branch unit has been duplicated on the figure 2 die, no inter-unit predicate propagation is needed and the 'p' path may be removed.

4 ISA Typing

To allow a full separation of the computing machines, it is necessary to give each some instruction addresses computing facilities (branches and jumps targets computations) as well as some data addresses computing ones (data structures pointers). In such a way, a function can be fully handled by its hosting unit. For example, a floating point function such as SAXPY, that is made of floating point computation instructions, load and store ones and loop control ones, is entirely taken in charge by the fpu. This means that the fpu register file keeps the floating point arrays pointers and index as well as the floating point data. It also induces that the set of functional units includes an integer adder to handle address computations. The instruction set itself is enhanced to give to each computing machine proper branching, jumping, loading and storing instructions (with an address computation based on pointers locally available in the machine register file).

The following piece of code shows what SAXPY looks like when written with such an instruction set. As we can see, every instruction is prefixed with 'f'. This designates the fpu computing unit to be used to compute the function. The opcode suffix gives the operation to be performed that can be a true floating point one as in 'fmul' and 'fadd', a load or a store of a floating point data ('fld' and 'fst'), the computation of an integer value ('ficlr' and 'fiadd') or a comparison of integers in a conditional branch ('fibtrue'). The return instruction itself is typed, to distinguish a floating point function return from an integer one. The call instruction is also typed with the prefix giving both the caller and the callee types (for example, the fpu has three calls: 'ficall', 'ffcall' and 'fgcall' to call respectively a floating point, an integer or a graphic function). To handle calls and returns as well as predictions, two hardware stacks per unit are maintained. One, say RS, is the return address stack (as in actual speculative processors) and the other, say CS, is a continuing unit stack.

```
saxpy:      /*x:f1; y:f2; a:f3; n:f4*/
            /*for (i=0;i<n;i++)*/
            /*    y[i]+=a*x[i];*/
            ficlr f16            /*i=0*/
e0:         fld f17=M[f1+f16*4]  /*x[i]*/
            fmul f17=f17*f3      /*a*x[i]*/
            fld f18=M[f2+f16*4]  /*y[i]*/
            fadd f17=f17+f18     /*y[i]+=a*x[i]*/
            fst M[f2+f16*4]=f17  /*y[i]*/
            fiadd f16=f16+1      /*i++*/
            fibtrue (f16!=f4),e0 /*loop*/
            fret
```

```
        ...            ;fpu is the active unit
        ficall  max    ;push fpu on top of CS(iu)
                       ;push "a" on top of RS(fpu)
a:      ...            ;fpu is the active unit
        ...            ;fpu is the active unit

max:    ...            ;iu is the active unit
        ...            ;iu is the active unit
        iret           ;pop CS(iu); pop RS(fpu)


        u...           ;u is the active unit
        u...           ;u is the active unit
        u...           ;u is the active unit
        u'...          ;trap; u' becomes the active unit
        u'...          u' is the active unit
        u'...          u' is the active unit
        u'...          u' is the active unit
        u...           ;trap; u becomes the active unit
        u...           ;u is the active unit
```

Fig. 3. **Typed function call and return**

A unit is made active when a call of its type is fetched (see figure 3 upper part; for example, if the fpu fetches a 'ficall', the integer unit is made active). The previous active unit code is pushed on top of CS (in the example, the fpu code is pushed on top of the integer unit CS stack; the return address is pushed on top of the fpu RS stack). The same unit remains active until the return instruction is fetched (in the example, a return from an integer function; in a return instruction, the target function type is not mentioned to allow returns to different function types). While a unit is active, the other ones are inactive (hence, units read one at a time in the instruction and data L2 caches). When a return is executed, the leaving unit pops the continuing unit from its CS stack. The leaving unit is disactivated (its fetch unit no more fetches) and the continuing unit is reactivated (its fetch unit pops the return address from its local prediction stack RS and restarts fetching). Disactivation and reactivation are performed within the same cycle, incurring no delay to switch the active unit.

Branches and jumps are predicted the same way than in a speculative processor (with one prediction subunit per type). A bad prediction correction is internal to a unit except for calls and returns. When a call is mispredicted, the calling unit resets the called one with the corrected address. When a return is mispredicted, first the falsely predicted continuing unit is disactivated and second, the true return address designates the true continuing unit (such a distinction requires that the OS separates the code region in typed segments; an alternative is to trap at fetch when a type t' instruction is fetched by a type t unit; this is further explained later).

Such a typed coding highly relies on the compiler ability to recognize each function type. For example, in C, the SAXPY function would have the following prototype:

```
void saxpy(float x[], float y[], float a, int n)
```

This example shows that the function type itself (void) does not give the computing unit type. Moreover, the set of arguments is mixed. However, in this example the compiler should easily decide to match the function with the fpu. Some other cases might be more complicate for example when the function computes separate values of different types. Then, the compiler has to analyse the data dependencies to separate the function code into properly typed subfunctions. The connection between the subfunctions can be realized with a typed call instruction or by trapping. With the trapping technique, when a unit u (see figure 3 lower part) fetches an instruction belonging to unit u', a trap occurs that switches the active unit from u to u' (as if a typed call would have been fetched). This kind of trap occurs at fetch time. It incurs a one cycle delay (a block is fetched and a badly typed instruction is detected in it; it disactivates the active unit (u) and reactivates the continuing unit (u') with the trapping instruction address for pc; the instruction is refetched by the unit of its type (u')). The compiler should reorganize the code to minimize such types breaks (we must insist on the fact that mixing pointers and data does not lead to types breaks; as in the SAXPY example above, we can see that floating point pointers and indexes are handled by the fpu with its register file; for this reason, types breaks should be rare in the dynamic code run from our proposed ISA; as an example, the SAXPY code does not contain any type break).

The case of data is different. A function accesses data memory through its unit private cache. For example, a function handled by the fpu accesses floating point data and pointers on floating point structures in the fpu data cache. If a datum must be shared by two units, an explicit conversion must be performed that is coded with an import instruction that moves a register from an external unit to a register of the local unit. For example, this is the case in a conversion function. The function (see an example on figure 4 with the function $atof$) computes the conversion and leaves the converted value in one of its data registers. The caller (function $fpfun$) imports this register into its proper file (instruction $fimport$). The computing units are linked by a bus that only serves for this purpose. In such a way, a datum is typed and never has to be loaded for update in the cache of another type (it can be loaded as a side effect with other data sharing the same cache line in another cache but cannot be modified; for example, a structure composed of an integer and a floating point datum can be loaded in the integer unit data cache as well as in the fpu one; however, the integer, if handled by the integer unit cannot be modified by the fpu, nor can the floating point data be modified by the integer unit (see figure 5)). These restrictions ensure cache coherency without the need of snooping.

```
atof:   ...             ;read an ascii word
        ...             ;and convert to a
        ...             ;floating point data
        ...             ;atof is computed by
        ...             ;the integer unit
        iret            ;fp data in R0, integer register


fpfun:  ...
        ficall atof     ;call to the integer function atof
        fimport F0=R0   ;get the fp data
        ...             ;(inter-register files transfer)
```

Fig. 4. **Inter-units importation**

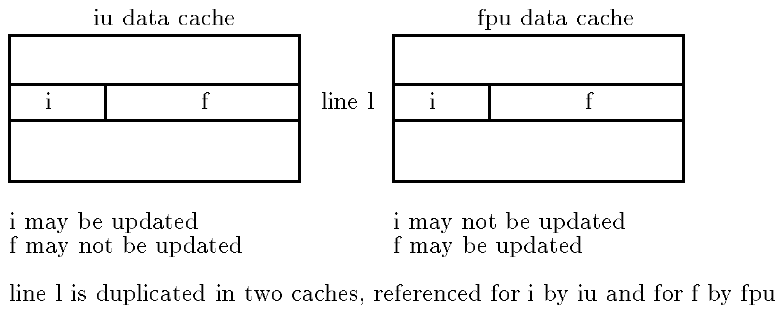

Fig. 5. **A duplicated cache line**

This typing of the computing units is compatible with any speculative and out-of-order execution mode as long as the techniques employed for implementation do not add any interunit communication except possibly some status signals.

5 Discussion

Different recent proposals have an impact on processor clustering. SMT-like multithreaded processors [14] have a very centralized organization, the different threads being merged into a unique shared core. This is an important drawback of SMT design that should lead to important inter-units communication delays. On-chip multiprocessors [9] cluster the die into independant processors that can communicate only through the shared part of their memory hierarchy (that can be on-chip or off-chip). The main disadvantage of this type of design comes from the resources redundancy imposed by the share-nothing choice and the inter-processors communications induced by the coherency protocol for the duplicated levels of the data memory hierarchy.

The Multicluster architecture [4], the Multiscalar architecture [13] and the Complexity-effective architecture [10] are examples of clustered single processor architectures. In each of them, the fetched instructions are dispatched to clusters. This is a first difference with this proposal. In these designs clustering does not concern the fetch unit, implying that the fetch latency does not get improved. A second difference is that we modify the ISA and they don't. For this reason, the cluster to which an instruction belongs is not opcode dependent (as it is in the present proposal) but is determined from two contradictory aims: avoid inter-cluster communications (this tends to centralize the instructions into a single cluster) and equilibrate cluster computation load (this tends to distribute instructions uniformly into clusters). Instructions forming a computation chain are oriented in the same cluster. Different chains are alternately placed in the available clusters. Instructions may have their sources in two different clusters implying an inter-cluster communication. This occurs often enough to impose one or more inter-cluster communication paths.

In fact their goal is not the same as our. They want to reduce the necessary core resources to allow a high superscalar degree (such as read and write ports on the register file). Each cluster takes in charge a subset of the superscalar degree (for example, a degree 16 machine could be composed of four degree 4 clusters). What we want is to decrease the number of inter-unit communications and to reduce the paths lengths. If a high superscalar degree is needed, then each computation unit has to be clustered itself (in the proposed design, each unit when it is active fetches as many instructions as the superscalar degree allows), which is out of the scope of this paper.

6 Conclusion

In this paper, we have first pointed out that inter-units communication paths and mainly fetch and load/store paths now impact processor performance. The fetch and the load/store paths can be shortened with a reduced size L1 cache. The L1 miss penalty can be reduced if the L1 and L2 caches are simultaneously accessed. We have also noted that even though the L1 cache is kept small to scale with the cpu cycle, the core area increases leading to an increase in the drive time along the paths connecting the caches to the computing units.

By typing the ISA, we have shown that it is possible to half the fetch path length. The instruction cache, the fetch unit, the branch prediction unit, the typed computing unit, the load/store unit and the data cache altogether form an independent machine that has quite no link with the other computing machines in the cpu, except inter-register import. During a cycle, a single machine is the active machine, i.e. the one that fetches. A machine is made active with a function call of its type, a function return or a fetch trap.

ISA typing should be useful to help to scale the processor cycle with the technology in future designs. With ISA typing, it is possible to reduce the pipelines depths by removing some or all of the drive stages. This means that the CPI can be better sustained despite the dramatic cycle reduction.

References

[1] Athlon: http://www.amd.com/products/cpg/athlon/pdf/architecture_wp.pdf

[2] W.J. Dally and J.W. Poulton: Digital systems engineering. CUP 1998.

[3] J. Edmondson et al.: Internal Organization of the Alpha 21164, a 300-MHz 64-bit Quad-issue CMOS RISC Microprocessor. Digital Technical Journal, vol. 7, no. 1: 119-135.

[4] K. Farkas et al.: The multicluster architecture. Reducing cycle time through partitioning. Micro30, 1997

[5] C.T. Gray, W. Liu and R.K. Cain III: Wave pipelining: theory and CMOS implementation. Kluwer academic publishers, Norwell, 1993.

[6] HP8500 ftp://www.hotchips.org/pub/hot7to11cd/hc98/pdf_1up/hc98_1a_johnson_1up.pdf

[7] IA64 http://developer.intel.com/design/ia-64/manuals

[8] Richard E. Kessler: The Alpha 21264 microprocessor. IEEE Micro, 19 (2), March-April 1999, 24-36.

[9] K. Olukotun, B. Nayfeh, L. Hammond, K. Wilson, K. Chang: The case for a single-chip multiprocessor. ASPLOS7, 1996.

[10] S. Palacharla and N. Jouppi: Complexity-effective superscalar processors. ISCA24, 1997

[11] Pentium III http://developer.intel.com/technology/itj/q21999/articles/art_2.htm

[12] Pentium IV ftp://download.intel.com/pentium4/download/netburstdetail.pdf

[13] K.K. Sundararaman, M. Franklin: Multiscalar execution along a single flow of control. ICPP 1997.

[14] D.M. Tullsen, S.J. Eggers, H.M. Levy: Simultaneous multithreading: maximizing on-chip parallelism. ISCA22, 1995.

[15] Steven J. E. Wilton and Norman P. Jouppi. An Enhanced Access and Cycle Time Model for On-Chip Caches. Compaq WRL Research Report 93/5, July 1994.

http://www.research.digital.com/wrl/techreports/abstracts/93.5.html

http://research.compaq.com/wrl/people/jouppi/cacti2.pdf