Functional Unit Sharing Between Stacked Processors in 3D Integrated Systems

Demid Borodin, Winston Siauw and Sorin Dan Cotofana
Computer Engineering Laboratory
Faculty of Electrical Engineering, Mathematics, and Computer Science
Delft University of Technology
Mekelweg 4, 2628 CD Delft, The Netherlands
Telephone: +31 152787362, Fax: +31 15 2784898.
E-mail: D.Borodin@TUDelft.nl

***Abstract*—The emerging Through Silicon Via (TSV) based 3D integration technology provides the means to stack two or more dies, enabling a low-latency interface between them. Apart of the immediate advantages of such an approach, e.g., short wires, it also opens research avenues for 3D organizations of computation platforms. In this line of reasoning we propose in this paper to share resources between stacked processors while focusing on Functional Units (FUs) inter-die sharing. The purpose of FU sharing is two fold: (i) it enables inexpensive fault tolerance by allowing, when possible, redundant instruction execution on idle FUs of processors located on other stack dies; and (ii) it can result in performance improvements by remotely executing instructions on idle FUs located on other dies in the 3D stack, when more instructions than locally available FUs are issuable. We evaluated the potential implications of our proposal on a 3D system built with two stacked processors (dies). In this case only a limited error detection capability is enabled and the experimental results indicate that our scheme covers on average 46% of the executed instructions. When performance improvement is targeted an average speedup of 6.9% can be achieved for the applications running on the considered two die stack.**

I. Introduction

Modern semiconductor industry faces severe scaling problems while attempting to keep pace with Moore's Law [1]. In particular, one of the major challenges is the increasing communication delay [2] due to long interconnect wires. Signal propagation delay and energy consumption of long wires do not scale so well as the transistor size does. The emerging three-dimensional (3D) integration technology [3], [4] has the potential to alleviate some of these problems as it significantly contributes to overcoming the interconnect scaling barriers, by providing designers with the third dimension. The third dimension enables them to replace long intra-die interconnect wires with significantly shorter inter-die (vertical) communication channels implemented with Through-Silicon Vias (TSVs). By making use of the z-dimension the 2D circuits can be folded at different granularities, such as stacking independent processors, functional units, or splitting a functional unit across different dies [3]. Thus, 3D staked implementation of any wire-dominated circuit can significantly outperform the 2D counterparts in terms of performance [5] and energy consumption [6].

This work was partially supported by the Dutch Ministry of Economic Affairs in the context of the 3DIM^3 CATRENE project #CT105.

In addition to the straightforward benefits achievable by replacing long wires with shorter ones in existing systems, 3D integration opens new design opportunities. These opportunities can be leveraged to even further improve the system. For example, dies produced with different fabrication technologies can be stacked together. This enables, for instance, to stack high-density DRAM dies with high-performance CMOS process [7] (known as heterogeneous integration). As another example, the low-latency vertical communication channels between dies with processors and dies with caches allow for the placement of cache banks closer to the processors which make use of them, and thus reduce the cache access time [8].

In this paper we introduce a novel resource sharing technique applicable to 3D stacked computation platforms. Our method relies on the utilization of a TSV based low-latency interface between stacked dies to enable a fast mutual access between different parts of processors residing on different dies. In this way processors can get access to each other's resources and in the case that some resources are underutilized on processor A while processor B is in shortage of such resources, it can utilize the available remote resources.

To evaluate this approach, we apply the resource sharing idea to processor functional units. Functional unit sharing between processors residing on stacked dies of a multiprocessor 3D integrated system can be utilized for two distinct purposes: (i) to improve the system reliability and (ii) to improve performance. Reliability improvement is achieved by introducing a partial fault detection (and possibly correction) capability in the instruction execution mechanism. If an idle functional unit of the appropriate type is available on the stacked processor B, the instruction being executed on processor A is also redundantly executed on that functional unit of processor B. The results of the original and the redundant executions are then compared in order to detect possible faults which do not affect both processors in the same way (so that processors A and B produce the same wrong result). This corresponds to duplication with comparison, a particular case of the classical N-Modular Redundancy fault detection scheme [9], [10]. For

redundantly executed instructions, this scheme detects both transient and permanent faults in functional units and data communication channels between register files, instruction window, and functional units. If required, other parts of the pipeline can be protected with data verification methods [11], such as parity and error detection/correction codes.

Performance improvement is achieved by remotely executing ready instructions on idle functional units of the stacked processors, when no appropriate resources are available on the issuing processor. In other words, if processor A has an instruction which is ready but cannot be issued due to the lack of available local functional units, this instruction is executed on a corresponding idle functional unit of processor B. This way, some stalls due to the lack of available functional units are eliminated by making use of the available resources in the 3D stack.

We evaluate both the reliability and performance aspects of the proposed functional unit sharing scheme using a simulator based on the SimpleScalar tool set [12]. Two processors stacked in a 3D system are simulated, each having a private memory and running independent workloads. The results indicate that when the system is configured to improve reliability, on average 46% of executed instructions can be protected (redundantly executed on another processor) without affecting the execution time. When performance improvement is targeted, an average speedup of 6.9% can be achieved.

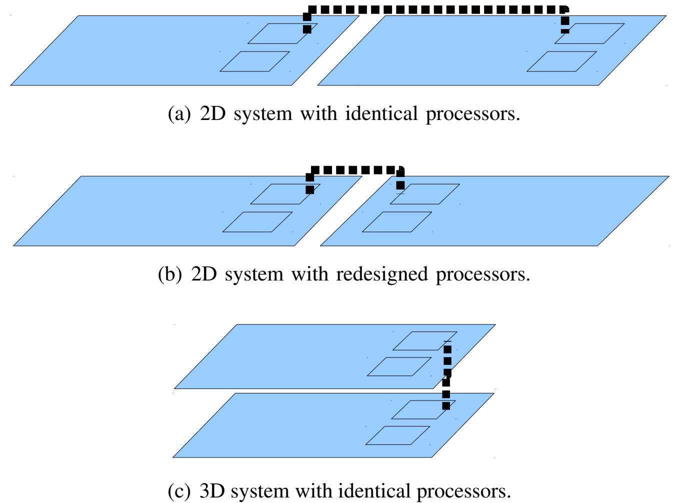
The remainder of this paper is structured as follows. Section II motivates and describes the implementation of the proposed functional unit sharing mechanism. Section III presents the experimental results evaluating functional unit sharing from the reliability and performance points of view. Finally, Section IV draws conclusions and describes some future work directions.

II. Functional Unit Sharing

This section starts by explaining the rationale behind resource sharing between stacked processors in 3D multiprocessor systems (Section II-A). Subsequently it presents the proposed 3D system organization which supports functional unit sharing (Section II-B). Finally, it explains how functional unit sharing can be used to improve reliability (Section II-C) and performance (Section II-D) of individual processors, and thus, of the entire 3D computation platform.

A. Motivation

The lack of resources is a common issue faced by processors during code execution. Some examples are insufficient capacity of structures (such as integrated fast cache, branch history table, instruction window etc.) and insufficient number of computational resources of the required type (such as functional units). The lack of resources can be solved (to a certain extent) by enlarging these resources on the processor, or increasing their number, at the expense of area and power overhead. The disadvantage of this approach is especially prominent if the resources are underutilized, which is quite frequently the case when applications being executed feature insufficient parallelism. Underutilized resources still consume power (possibly less than active resources), and the more of them are present, the more significant the waste is. However, in a multiprocessor system, it is quite probable that one processor is short on a certain type of resources, while another processor has them being idle. Instead of adding resources to every processor to solve the resources shortage problem (and by implication to increase the resource waste when underutilized), it would be beneficial if individual processors could share their (idle) resources with other processors in the system.



(a) 2D system with identical processors.

(b) 2D system with redesigned processors.

(c) 3D system with identical processors.

Fig. 1. Resource sharing in 2D and 3D systems.

Figure 1 depicts possible ways to implement resource sharing between processors. Unfortunately, the feasibility of resource sharing between processors in traditional two-dimensional (2D) systems is very doubtful. This is due to the fact that to provide a direct communication between corresponding resources in different processors, large numbers of very long wires must be introduced, as indicated in Figure 1(a). To avoid very long wires, it is possible to redesign the processors so that the shared resources are placed on the neighbor edges (which is possible if only a few resources are shared), as Figure 1(b) suggests. However, this approaches requires redesigning the layout of the entire processor, and it is not scalable as only two processors can implement resource sharing this way. The emerging 3D integration technology can solve these problems as it enables resource sharing between similar stacked processors (avoiding significant redesign), using short vertical communication channels (such as TSVs), as depicted in Figure 1(c). The 3D approach is scalable: it allows stacking of multiple processors, as long as the communication overhead does not become prohibitive.

This work focuses on sharing Functional Units (FUs) between stacked processors. Depending on the workload being executed, on its available Instruction-Level Parallelism (ILP) and Task-Level Parallelism (TLP), it is quite likely that some

processors in a system suffer from functional units shortage, while others might have idle units due to stalls. However, other processor elements can also be considered for such sharing. For example, it might be useful to share branch predictors if processors are running the same application. Depending on the achieved access time, it might also be useful to share L1 caches.

We suggest that functional unit sharing can be employed to improve two system aspects: reliability and performance. As a side-effect of performance improvement, a total system energy consumption reduction can be expected, because the execution time is shortened. The following sections introduce the system organization, and discuss how functional unit sharing can be utilized in order to improve reliability and performance.

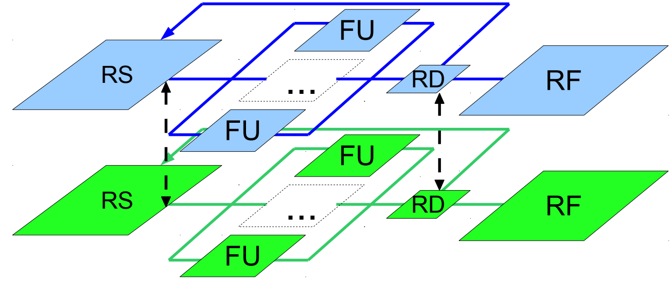


Fig. 2. System employing functional unit sharing. *RS*=reservation station, *FU*=functional unit, *RD*=results distributor, *RF*=register file.

B. Organization

Figure 2 presents the organization of a 3D system with stacked processors employing the proposed functional unit sharing mechanism. Only two processors are presented for the sake of simplicity. Ready instructions residing in Reservation Stations (RS) of any of the stacked processors can be issued both to local and to remote functional units. This organization can be seen as a set of stacked processors sharing a single pool of functional units.

The way processors interact in order to determine the functional units availability and decide where to remotely execute available instructions is a key issue in the design of the functional unit sharing scheme. In this paper, however, we do not address this aspect in details as we are mostly interested in determining the potential impact our proposal might have on the 3D system performance. In view of that, we assume for the sake of simplicity, that when a processor has an instruction to execute remotely, it broadcasts it (along with its input operands and the issuing processor's identifier) to all the stacked processors. Processors with idle functional unit(s) of appropriate type send an acknowledgement and execute this instruction. If at least one acknowledgement has been received, the issuing processor knows that the instruction is being remotely executed. After execution, the result is delivered back to the issuing processor by the Result Distribution (RD) unit. We note that other sharing mechanism can be used and we are currently investigating this issue.

The complexity of the network required for the inter-die communication depends on the number of processors involved in resource sharing. For the case with only a few stacked processors, a simple bus is sufficient. A more complex scalable network might be desirable for large numbers of communicating processors. Alternatively, the processors can form multiple sharing groups communicating through simple buses. This is, however, outside the scope of this work, since stacking large numbers of dies in 3D integrated systems appears not to be feasible in foreseeable future.

C. Reliability Mode

With a given technology featuring fixed characteristics, reliability is usually improved by introducing some type of redundancy. This can be space (additional hardware), information (additional data) and/or time (performing an operation multiple times sequentially) redundancy [9]. For very critical systems such as those used in aviation and for military purposes, significant amounts of expensive space redundancy is justified. It is common, for example, to run the same workload on multiple processors and make sure their results match on those systems. For the desktop and embedded markets, however, the demands significantly differ. On one hand, modern technology trends such as shrinking the feature size and increasing the integration level have increased the fault probability [13], leading to the need to integrate fault tolerance features. On the other hand, very expensive fault tolerance methods are prohibitive for these markets. This led to an extensive research in the area of inexpensive fault tolerance approaches.

One such fault tolerance approach duplicates the executed instructions and compares their results, making sure they match. This corresponds to the classical duplication with comparison error detection method, a special case of the N-Modular Redundancy fault detection and/or correction scheme [9], [10] with N equal to two. Instructions can be duplicated both in hardware [14] and in software [15]. Instruction duplication in hardware can also take the form of thread duplication [16] based on Simultaneous Multithreading [17]. It has been indicated that a partial fault coverage (protecting only a subset of executed instructions, when required resources are available) is still beneficial [18].

In this work we use functional unit sharing between stacked processors to implement a partial fault detection at a very low cost. Fault detection is achieved by instruction duplication and results comparison in hardware. When executing an instruction, the processor looks for an idle functional unit of the appropriate type on the other stacked processors. If one is found, a redundant copy of the instruction is executed there, and the results are compared to detect possible faults. This way, without using dedicated redundancy, when resources are idle on other processors, they are used to improve reliability. Note that as an additional benefit, this organization protects against possible common faults better than executing on a dedicated local redundant functional unit. This is because a local functional unit on the same die is more likely to be

affected by the fault source in the same way as the original unit, and thus it is more likely to produce the same wrong result.

As mentioned earlier, only idle functional units are used for redundant execution. If the execution takes only one clock cycle, or if the functional unit is pipelined, this never affects the performance of the processor whose unit is shared. The only case when this scheme can degrade performance of the sharing processor is if a multi-cycle operation is executed on a functional unit which is not pipelined, and which must be used by the local processor before the remote operation completes.

D. Performance Mode

Running applications often face a lack of functional units. This happens at high-ILP phases, when the number of ready instructions exceeds the number of available appropriate functional units. To solve this problem, functional unit sharing is used in this work to employ other processors' functional units, if possible.

When a processor is unable to issue an instruction because all the appropriate local functional units are busy, it looks for an idle functional unit of appropriate type on the other stacked processors. If it finds one, the instruction is executed there, as if that functional unit were local. This way, processors in low-ILP phase help processors in high-ILP phase to avoid stalls.

Two strategies are possible when implementing the functional unit sharing in performance mode. The issuing processor can try to execute an instruction locally, and if it does not succeed, try to execute it remotely immediately. Alternatively, the issuing processor can first try to issue as many instructions as possible locally. Then, if still possible, it tries to execute some instructions on remote functional units. In this work the second approach is chosen, because it provides the following advantages. First, because the local execution has a priority, fewer instructions are executed remotely. This decreases the amount of traffic between the processors, and thus improves the scalability of the approach (more processors can be involved in sharing). Second, the synchronization between the processors is expected to be simpler: because all the local instructions are executed first, when a remote execution is attempted, the remote functional units are already likely to know if they are idle at this time.

As in reliability mode discussed in Section II-C, only idle functional units are shared in performance mode. Thus, it can only slow down the sharing processor if a multi-cycle operation is executed on a functional unit which is not pipelined, and which must be used by the local processor before the remote operation completes.

III. Experimental Evaluation

This section experimentally evaluates the proposed functional unit sharing methods. Section III-A describes the used simulator and benchmarks. Section III-B assesses the fault detection capability of functional unit sharing. Section III-C evaluates the execution speed benefits achieved in performance mode.

TABLE I
Base processor configuration.

Issue	out-of-order
Fetch/Dec./Issue Width	4
# of Int. ALUs	4
# of Int. Mult./Div.	1
# of FP ALUs	1
# of FP Mult./Div.	1
RUU Size	64
LSQ Size	32
Memory Latency	112 cycles (first chunk), 2 cycles (subsequent chunks)
L1 Data Cache	32 KB, 2-way set associative
L1 Instruction Cache	32 KB, 2-way set associative
L2 Unified Cache	512 KB, 4-way set associative

A. Experimental Setup

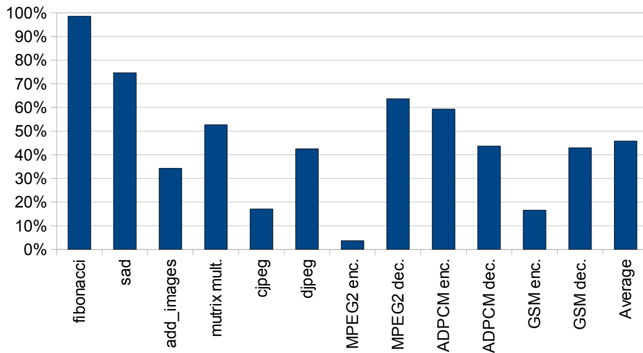
The evaluation is performed using a modified version of the SimpleScalar tool set [12]. The most detailed simulator, *sim-outorder*, has been adapted to model a 3D system, such that it can simulate multiple stacked processors employing functional unit sharing in either reliability or performance mode. The stacked processors have the same configuration and run different workloads. The system does not feature a shared memory, the processors work independently. The base configuration of every processor is described in Table I. The number of stacked processors in this work is limited to two.

In performance mode, various configurations are used to investigate how they affect the overall system performance. Both in-order and out-of-order execution models have been explored. In addition, the number of integer ALUs has been varied from 1 to 4, and the fetch, decode, issue, and commit width changed accordingly to preserve a balanced organization. We only focus on integer ALUs because integer arithmetic operations significantly dominate (constitute about 90%) in the considered benchmarks.

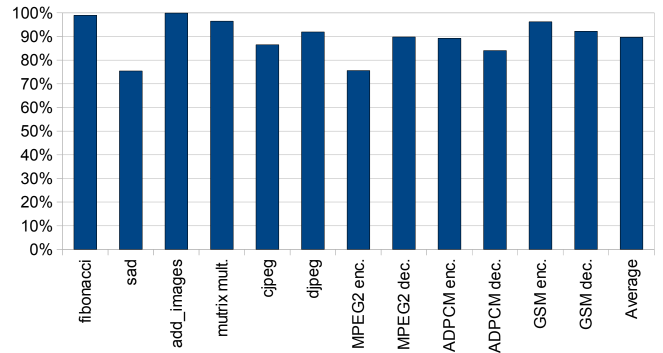
Since multimedia domain is one of those targeted by 3D architectures, several multimedia kernels and applications are used as benchmarks for the evaluation. Image Addition, Matrix Multiplication (with the input matrices of the size 200×100 and 100×200), and Sum of Absolute Differences (SAD) are kernels very often used in multimedia applications. The fourth kernel computes the Fibonacci numbers, which are widely used in science, and even in financial market trading and music [19]. As full applications, encoders and decoders for the following standards (taken from the MediaBench benchmark suite [20]) are used:

- JPEG image compression.
- MPEG2 video compression.
- ADPCM sound compression.
- GSM sound (voice) compression.

Unless specifically mentioned otherwise, functional unit sharing between stacked processors is only enabled while both processors are busy executing some workloads. If one processor finishes executing earlier than the other, the sharing



(a) Strict case (functional unit sharing on idle processors is disabled).



(b) Ideal case (one processor is idle).

Fig. 3. Percentage of instructions executed redundantly using functional unit sharing.

is disabled. This is because all functional units on the idle processor are free, and thus they will fulfill any request from the other processor. In real multiprocessor systems it can happen that some processors are idle if no sufficient TLP is available. However, in this study we explore the worst case when all processors are busy. To avoid significant influence on the results, we have tried to couple (run on different processors) benchmarks with comparable execution times. However, in several cases one benchmark still completes significantly earlier than the other one, thus disabling the functional unit sharing for the rest of that benchmark, and worsening the results.

B. Reliability Mode

Figure 3(a) depicts the percentage of instructions executed redundantly on another processor using functional unit sharing in reliability mode. Here and subsequently, the benchmarks presented in figures are executed in pairs (on two stacked processors) in the given order. For example, *fibonacci* is executed on one processor, and *sad* on the other. For another experiment, *add_images* is executed in parallel with *mutrix mult.*, etc.

Figure 3(a) suggests very different results for various benchmarks. The best results are achieved when both processors execute low-ILP workloads, such as *fibonacci* and *sad*, which represent a sequence of dependent operations. When executing a low-ILP workload, a processor executes fewer instructions in parallel, and thus sends fewer instructions to other processors to re-execute. At the same time, the processor has more idle functional units that are available for sharing. On the contrary, high-ILP workloads generate many instructions to execute in parallel, saturating the local processor, and submitting many instructions for remote execution.

Figure 3(a) indicates that on average 45.8% of executed instructions are protected in reliability mode. Note that the functional unit sharing is only enabled when both stacked processors are busy, this in some cases negatively affects the results. Moreover, as mentioned above, it can happen that some processors in a multiprocessor system are idle due to the lack of TLP. In this case, all their functional units are available for sharing. To explore also the best-case scenario, Figure 3(b) presents the case when one of the two processors in the system is always idle, case in which on average 89.7% of instructions are executed redundantly. In reality, depending on the TLP available in the system workload, the results can be expected to fit in between those in Figure 3(a) and in Figure 3(b).

Note that Figure 3(b) indicates that not all instructions are executed redundantly, although all the functional units on the other processor are available. This is due to the fact that in our setup not all types of instructions can be shared. In the absence of a shared memory, memory access instructions cannot be shared, because processors have their private memory spaces and cannot access each other's.

C. Performance Mode

Figure 4 presents the speedup achieved in the performance mode over the normal execution without functional unit sharing. Different processor organizations are explored, with both in-order and out-of-order execution models. Functional unit sharing in performance mode does not prove valuable when the stacked processors have balanced organizations. With 4 integer ALUs (with the corresponding fetch/decode/issue/commit width of 4) the speedup varies from 0% to 0.4% in out-of-order system, and barely exceeds 0% in in-order system. However, for unbalanced processor organizations (with either reduced number of functional units or increased fetch/decode/issue/commit width), significantly better results are achieved.

To explain this behavior, Figure 5 presents the number of candidate instructions, and Figure 6 presents the hit rate. Candidates are instructions that do not find an appropriate functional unit on the local processor, and try to execute remotely. Hit rate represents the percentage of candidate instructions that succeed to execute remotely. Note that (as shown in Figure 5) there can be more candidate than executed instructions. This happens when the system resources are so saturated that instructions become candidates multiple times. If an instruction does not find a functional unit neither on the local nor on a remote processor, it can become a candidate again on the next clock cycle.

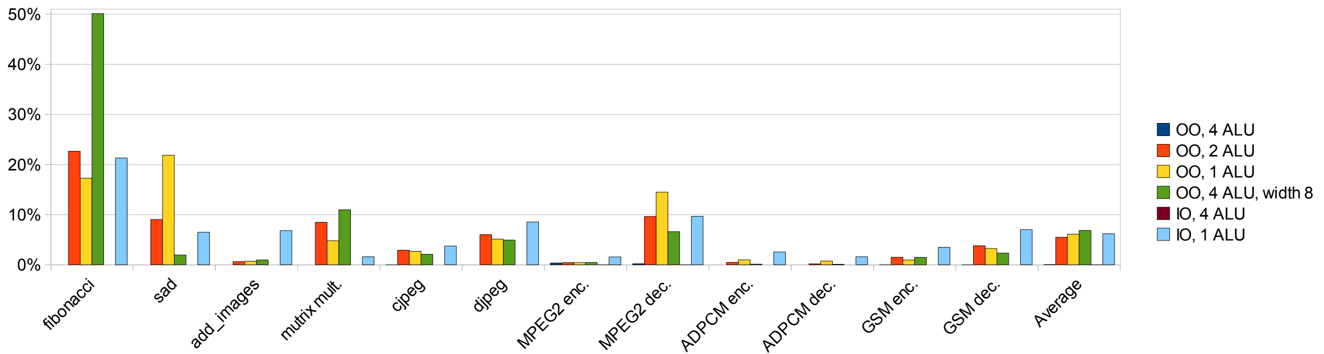


Fig. 4. Speedup in performance mode over normal execution without functional unit sharing. *OO*=out-of-order, *IO*=in-order, *width*=fetch/decode/issue/commit width.

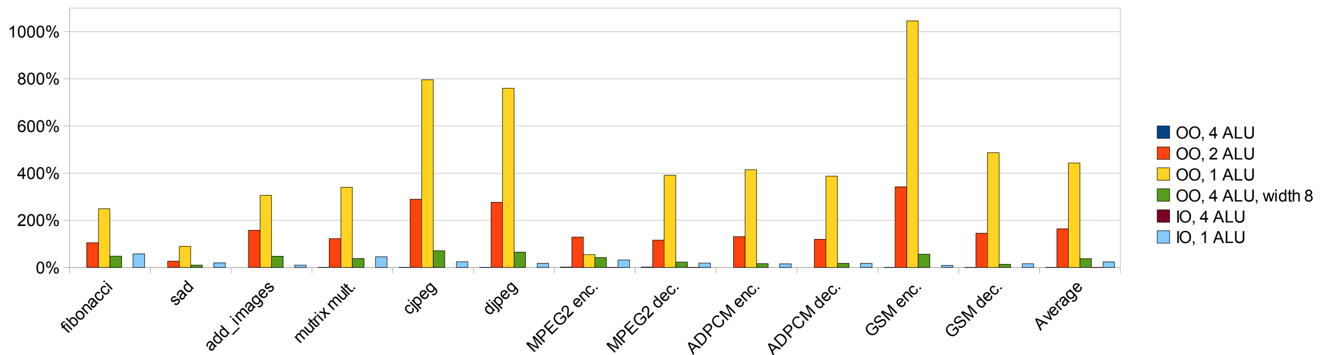


Fig. 5. Ratio of candidate instructions to the number of executed instructions.

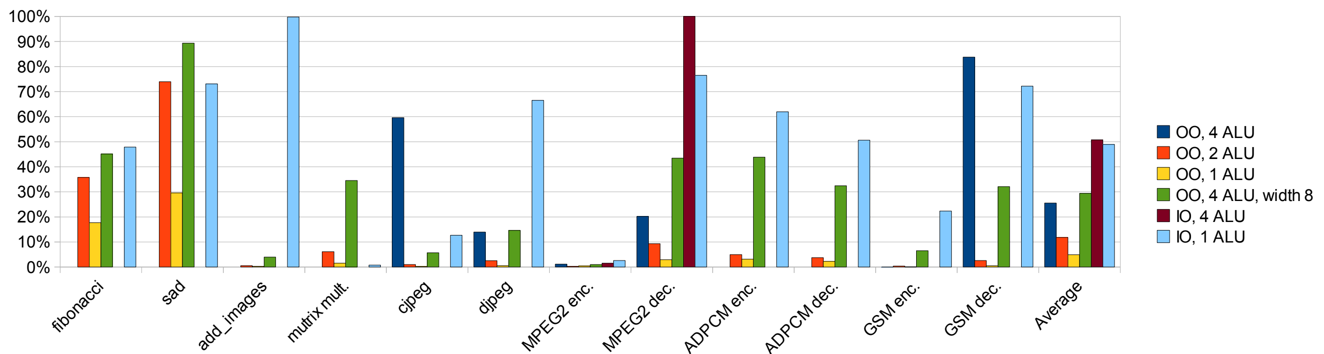


Fig. 6. Percentage of candidate instructions that hit (execute on a remote functional unit).

Figure 5 indicates that on balanced systems, there are very few candidates to execute on another processor (up to 1.9% of executed instructions). This explains why balanced organizations achieve almost no speedup as although a large percentage of candidates becomes hits (see Figure 6), there are too few instructions to execute remotely. With fewer functional units, the candidates ratio grows, achieving almost 8 times the number of executed instructions with a single integer ALU. However, with fewer functional units, fewer of them are available on remote processors, and thus less percentage of candidates hit. This is why the average speedup on processors with one ALU does not significantly exceed the speedup with two ALUs.

Based on these observations we can deduce that the recommended approach is not to reduce the number of integer ALUs, but to increase the fetch/decode/issue/commit width instead, to create more candidate instructions. Such a configuration generates on average 37.4% candidates (compared to 0.3% in a balanced out-of-order system), out of which on average 29.4% hit. The average speedup achieved with this approach is 6.9%. The average speedup achieved by the best another configuration (with one integer ALU) is 6.1%. Moreover, the

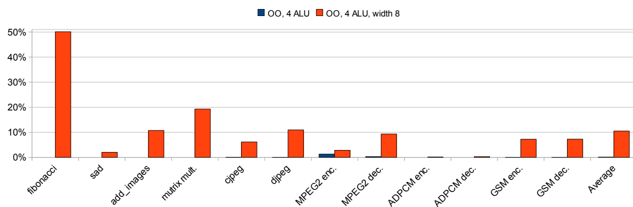


Fig. 7. Speedup in performance mode over normal execution. Ideal case when one of the stacked processors is idle.

absolute execution speed in the recommended case is 2.1 times faster than the one ALU case.

To explore the limits, the ideal case (when one of the stacked processors is idle) has been investigated for the base and recommended configurations. Figure 7 presents the speedup results. As mentioned above, in reality speedups in between Figure 4 and Figure 7 can be expected. Due to moderate numbers of candidate instructions, the average speedup of a balanced system is still negligible, and that of a system with enlarged fetch/decode/issue/commit width is 10.6%.

IV. Conclusions

This work proposes resource sharing, focusing on functional units, between processors stacked in 3D integrated multiprocessor systems. Functional unit sharing is used to improve the system reliability or performance. In reliability mode, it enables inexpensive fault detection capability without introducing significant hardware redundancy and without affecting performance. In performance mode, it speeds up execution without introducing significant overhead. Experimental results demonstrate that on a system with two stacked processors, functional unit sharing in reliability mode is able to protect (in the worst case) on average 45.8% of executed instructions. In performance mode, it is able to reduce the execution time by 6.9% for processors with enlarged fetch/decode/issue/commit width. As a positive side effect, performance improvement is expected to reduce the system energy consumption.

The hardware cost of functional unit sharing implemented using TSVs is expected to be low, if only a limited number of processors are involved in sharing. A more detailed investigation of this aspect is planned as a future work. In particular, we plan to determine the maximum feasible number of processors preventing that the sharing communication becomes a system bottleneck. We also plan to evaluate functional unit sharing between more than two stacked processors. In addition, we plan to evaluate if using combinations of reliability and performance modes can further improve the results.

References

[1] G. E. Moore, "Cramming More Components Onto Integrated Circuits," *Electronics*, vol. 38, no. 8, pp. 114–117, April 1965.

[2] R. Ronen, S. Member, A. Mendelson, K. Lai, S. lien Lu, F. Pollack, John, and J. P. Shen, "Coming Challenges in Microarchitecture and Architecture," in *Proc. IEEE*, 2001, pp. 325–340.

[3] Y. Xie, G. H. Loh, B. Black, and K. Bernstein, "Design Space Exploration for 3D Architectures," *J. Emerg. Technol. Comput. Syst.*, vol. 2, no. 2, pp. 65–103, 2006.

[4] B. Black, M. Annavaram, N. Brekelbaum, J. DeVale, L. Jiang, G. H. Loh, D. McCaule, P. Morrow, D. W. Nelson, D. Pantuso, P. Reed, J. Rupley, S. Shankar, J. Shen, and C. Webb, "Die Stacking (3D) Microarchitecture," in *MICRO-39: Proc. of the 39th Annual IEEE/ACM Int. Symp. on Microarchitecture*. Washington, DC, USA: IEEE Computer Society, 2006, pp. 469–479.

[5] A. Rahman and R. Reif, "System-Level Performance Evaluation of Three-Dimensional Integrated Circuits," *IEEE Trans. on Very Large Scale Integration Systems*, vol. 8, no. 6, pp. 671–678, 2000.

[6] J. W. Joyner and J. D. Meindl, "Opportunities for Reduced Power Dissipation Using Three-Dimensional Integration," in *Proc. of the IEEE 2002 Int. Interconnect Technology Conf.*, Burlingame, CA, USA, Jun 2002, pp. 148–150.

[7] G. H. Loh, Y. Xie, and B. Black, "Processor Design in 3D Die-Stacking Technologies," *IEEE Micro*, vol. 27, no. 3, pp. 31–48, 2007.

[8] F. Li, C. Nicopoulos, T. Richardson, Y. Xie, V. Narayanan, and M. Kandemir, "Design and Management of 3D Chip Multiprocessors Using Network-in-Memory," *SIGARCH Comput. Archit. News*, vol. 34, no. 2, pp. 130–141, 2006.

[9] B. Johnson, *Design and Analysis of Fault-Tolerant Digital Systems*. Addison-Wesley, Jan 1989.

[10] D. Siewiorek and R. Swarz, *Reliable Computer Systems: Design and Evaluation*, 3rd ed. A K Peters Ltd, Oct 1998.

[11] M. Franklin, "Incorporating Fault Tolerance in Superscalar Processors," in *HiPC-96: Proc. Third Int. Conf. on High-Performance Computing*. Washington, DC, USA: IEEE Computer Society, Dec 1996, pp. 301–306.

[12] T. Austin, E. Larson, and D. Ernst, "SimpleScalar: An Infrastructure for Computer System Modeling," *IEEE Computer*, vol. 35, no. 2, pp. 59–67, 2002.

[13] P. I. Rubinfeld, "Managing Problems at High Speed," *IEEE Computer*, vol. 31, no. 1, pp. 47–48, 1998.

[14] M. Franklin, "A Study of Time Redundant Fault Tolerance Techniques for Superscalar Processors," *Proc. IEEE Int. Workshop on Defect and Fault Tolerance in VLSI Systems*, pp. 207–215, Nov 1995.

[15] N. Oh, P. Shirvani, and E. McCluskey, "Error Detection by Duplicated Instructions in Super-Scalar Processors," *IEEE Transactions on Reliability*, vol. 51, no. 1, pp. 63–75, Mar 2002.

[16] E. Rotenberg, "AR-SMT: A Microarchitectural Approach to Fault Tolerance in Microprocessors." in *FTCS-29*, Madison, Wisconsin, USA, Jun 1999, pp. 84–91.

[17] D. Tullsen, S. Eggers, and H. Levy, "Simultaneous Multithreading: Maximizing On-Chip Parallelism," in *ISCA-95: Proc. 22nd Annual Int. Symp. on Computer architecture*, New York, NY, USA, 1995, pp. 392–403.

[18] M. Gomaa and T. Vijaykumar, "Opportunistic Transient Fault Detection," *ISCA-05: Proc. 32nd Annual Int. Symp. on Computer Architecture*, pp. 172–183, Jun 2005.

[19] "Fibonacci numbers at Wikipedia," http://en.wikipedia.org/wiki/Fibonacci_number.

[20] C. Lee, M. Potkonjak, and W. H. Mangione-Smith, "MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communicatons Systems," in *MICRO-30: Proc. of the 30th Annual ACM/IEEE Int. Symp. on Microarchitecture*. Washington, DC, USA: IEEE Computer Society, 1997, pp. 330–335.