FACTA UNIVERSITATIS (NIŠ)
Series: Electronics and Energetics vol. 10, No.2 (1997), 215-230

CATALYTIC MIGRATION: A STRATEGY FOR CREATION OF TECHNOLOGY–SENSITIVE MICROPROCESSOR ARCHITECTURES

Veljko Milutinović

Abstract. Existing approaches to the architecture, organization, and design of modern CPUs are summarized, and a promising new strategy is defined: catalytic migration. The discussion that follows concentrates on three issues: classification of catalytic migration approaches, examples of catalytic migration approaches, and the related performance evaluation.

1. Introduction

According to many researchers [e.g., 1], there are two basic approaches to supercomputer design. One is oriented towards extremely fast and powerful CPUs (of which just one or a few exist in the system). The other is oriented towards massive parallelism, using microprocessor technology (where the system is composed of a relatively large number of processing units).

Both approaches have merits, and the presentation which is the subject of this paper is applicable to both, although it yields better results in the case of the second approach. This is due to the fact that the strategy which is discussed here deals with technology–sensitive constraints on the type and quantity of resources that could be incorporated into the CPU. These constraints are "soft" (less stringent) if the design of a single but extremely fast and powerful CPU is not constrained to a single VLSI chip (constraints are

Manuscript received March 14, 1996.
Manuscript revised June 29, 1997.
This research was partially sponsored by the NCR World Headquarters, Dayton, Ohio, U.S.A. Simulation package ENDOT was donated by the TD Technologies Cleveland Heights, Ohio, U.S.A.

The author is with Faculty of Electrical Engineering, POB 35–54, 11120 Belgrade, Yugoslavia.

derived from the physical aspects of the design structure, cooling capabilities, interconnection delays, and similar). The constraints are "hard" (stringent) if the design of a single–chip CPU is considered, especially if GaAs implementation is involved (constraints are related to VLSI area limitations and the relatively large ratio of off–chip to on–chip delays). The strategy discussed here is more suitable for the environment with "hard" rather then "soft" limitations. Consequently, the rest of the text will be related to the "hard" environments, but the reader should be aware of the fact that the same conclusions (conditionally) also apply to the "soft" environments (environments with "soft" constraints).

This paper is predominatly oriented to technologies with a relative small on–chip transistor count, and a relatively large ratio of off–chip to on–chip delays. One example of such technology is GaAs.

The basic advantages of GaAs technology relate to relatively high speed, high radiation hardness, and robustness in harsh environmental conditions. All these issues are covered extensively in the open literature, and for details, the interested reader is referred to the existing references [e.g., 2,3,4, and 5]. The basic disadvantages of GaAs technology are its high production cost, relatively low yield which results in relatively small chips, and a relatively large ratio of off–chip to on–chip signal delays and/or memory access times. Again, these issues are assumed to be well known [e.g., 4 and 5], and will not be further elaborated here.

Numerous companies have been using GaAs as the implementational vehicle for their new computers. Supercomputer and supermini companies like Cray, ETA, and Gould have exploited the direction of the "random–logic" based design of extremely fast and powerful CPUs (using SSI, MSI, and LSI components), in conditions when the maximal attention has been dedicated towards advancing the state–of–the–art in the packaging and interconnection technology. On the other hand, companies like TI, McDonnell Douglas, and GE (RCA), for a number of years have exploited the direction of the VLSI based design of single–chip CPUs, in conditions when the maximal attention has been dedicated towards advancing the state–of–the–art in the GaAs semiconductor technology. More recently, one or the other approach has been adopted by some additional "large–scale" companies (e.g., AT&T, NCR, and Sun), as well as a few "start–up" companies (e.g., Gazelle Microcircuits, Pacific Semiconductor, and Prisma).

Concuriently, lots of good theoretical and practical research has been carried out in universities. Examples include, but are not limited to, University of Michigan [6], University of California in Santa Barbara [7], and Renselaer Polytechnic [8].

2. Problem statement

These days, the problem is not how to design and implement a 32–bit GaAs microprocessor. That problem has been successfully solved. What is the problem now is how to generate the architecture, organization, and design, so that a GaAs microprocessor is approximately N times faster than its silicon counterpart, where N is the speed ratio of the two technologies on the device level. What is relevant is the speed of the compiled high–level language (HLL) code, not the speed of the system clock[1].

In some of the existing 32–bit GaAs microprocessors, the clock is extremely fast. However, the memory pipelines of GaAs microprocessors tend to be fairly deep, due to the fact that the semiconductor technology has advanced much more than the packaging and interconnection technology. This makes it very difficult to handle branch delays, load delays, and coprocessor delays. Consequently, in most practical implementations, the speed of the compiled HLL code is not nearly as good as indicated by the clock speed. The ratio of the peak execution speed and the average execution speed tends to get fairly large [e.g., 9].

Therefore, the problem is how to achieve a speed–up which is close to N (as defined above). Importance of this problem is proportional to the overall manufacturing costs of GaAs microprocessors. In other words, the manufacturing costs are so high in GaAs that, unless the speed–up is good enough, the cost–performance product will not be as good as in silicon (of course, if other considerations like radiation–hardness dominate, the choice will be cost/performance independent).

The solution to the problem depends on the relationship between the semiconductor technology and the technology of packaging and interconnection. If advances in the packaging and interconnection technology follow the advances in semiconductor technology, the ratio of off–chip to on–chip delay will be relatively small (for GaAs, compared to silicon), and the fact that GaAs chips are relatively small will not require large changes in the architecture, organization, and design, compared to what is the current silicon practice. In other words, silicon solutions will work for GaAs as well (more on–chip registers and cache memory, or similar).

However it is believed that the advances in semiconductor technology will not be followed by corresponding advances in the packaging and interconnection technology. Therefore, the ratio of off–chip to on–chip delays will

[1]Fast system clock is sometimes achieved by making the pipeline deeper [9]. If applied compile–time optimization techniques are not efficient in the presence of deep pipelines, the speed of compiled HLL code can be very poor.

continue to be relatively large. Consequently, partitioning of functions across the boundaries of relatively small GaAs chips will continue to be an issue of major importance.

This paper assumes the latter scenario, and concentrates only on the solutions[2] which are aimed to an environment which is characterized by relatively small chips and relatively large ratios of off–chip to on–chip delays. Some researchers may argue that the packaging and interconnection technology can be substantially improved, and that the basic assumption of this paper will no longer hold (relatively high ratio of off–chip to on–chip delays). There are two answers. First, this design strategy is applicable to all technologies characterized with small chips and a relatively high ratio of off–chip to on–chip delays (today it is GaAs, tomorrow it is another technology). Second, systems have to be built today, before the more sophisbcated packaging and interconnection technology becomes available (this time period may last longer than expected). In conclusion, the strategy was born in the GaAs environment, but it should be treated in a technology–independent way[3].

3. Existing solutions

The approach to the architecture, organization, and design of "first generation" GaAs microprocessors was characterized by combination of "technology–related fine–tuning of the design" and "technology–related re–evaluation of the algorithms." Under the term "fine–tuning of the design" we mean adjusting the design parameters in existing solutions, so that the utilization of resources is not lowered when a new technology is applied. By "careful re–evaluation of the algorithms", we mean that whenever there exist more than one algorithm to implement a resource, all these algorithms should be re–evaluated, in order to find out which one is the best suited for the new technology (often, not the one which is traditionally believed to be the best, because it was the best under the conditions and criteria typical of silicon). Two representative examples follow, one for each of the two basic strategies defined above.

As for the "fine technology–related tuning," pipeline organization represents one good example. Typically, pipeline organization is determined through a trade–off between instruction fetch time and data path time. Elements of instruction fetch time are: (1) address propagation time, (2) memory access time, and (3) instruction propagation time. Elements of data

[2]This paper assumes that advances in GaAs technology are not followed by adequate advances in the packaging and interconnection tehnology.

[3]In the rest of the text, when we say GaAs, we mean GaAs or any other technology with the same or similar characteristics.

path time are: (1) concurrent reading of two operands from the register file, (2) signal propagation through the ALU, and (3) writing of the result into the register file. In silicon, for modern CPU designs with an on–chip instruction cache, the ratio of instruction fetch time to data path time is usually equal to one or smaller. Consequently, the elements of data path time are pipelined, which results in the so–called CPU pipelines of depth typically equal to 2,3, or 4. In GaAs, with an off–chip instruction cache, the ratio of instruction fetch time to data path time is almost always larger than one. Consequently, the elements of instruction fetch time are pipelined, which results in the so–called memory pipelines of depth typically equal to 2, 3 , or 4 (this does not say that instruction fetch pipelines cannot be found in silicon[4] microprocessors).

With regard to the "algorithmic re–evaluation," adder design represents a good example. There exist many ways to implement an adder [e.g., 11]. They range from the ripple–carry adder (which is treated as the slowest and the least complex to implement) to the carry–look ahead adder (which is treated as the fastest, and the most complex to implement), with a number of other solutions that fall in–between (according to the speed and the complexity of implementation). Actually, the above statements about the speed imply a performance evaluation model in which all gates have the same delay (which does not depend on the fan–in and fan–out of that particular gate), and that wire delays are negligible. These assumptions do not hold in GaAs! In a previous paper [12], we compared a number of different adders, using a model which reflects the differences between GaAs and silicon. It was found that, for word lengths of up to about 16 bits, ripple–carry adder may be the fastest. For larger word lengths, it will not be the fastest adder, but it may still be the most appropriate one to use in a 32–bit GaAs microprocessor, because it takes the least VLSI area. The saved VLSI area could be "invested" in the resources that have been proven useful in "fighting" with large off chip delays. In other words, the ranking of the "appropriateness" of different adders is different for GaAs (compared to silicon).

Solutions presented so far have enabled successful implementations of several 32–bit GaAs microprocessors on a single chip [13, 14, 9], but have not

[4]Note that computer systems of the 60's (semiconductor CPU and core memory) are also characterized with a large ratio of instruction fetch time and data path time, similarly as in the computer systems of the 80's (GaAs CPU and GaAs memory). However, there is one important difference: In the 60's, the memory access time (element #2) was much larger than the other two elements, address propagation time and function propagation time (elements #1 and #3). In the 80's it was the opposite. Consequently, solutions typically used in the 60's and the 80's are different (interleaved in the 60's and pipelined memory in the 80's).

contributed much to achieving the desired speed–up of N. Therefore, the researchers had to investigate other possible approaches[5]. One possible approach is presented next.

4. Catalytic migration

The approach proposed here is referred to as catalytic migration, and its essence is best explained through a comparison with the concept of direct migration. Therefore, direct migration will be explained first, followed by catalytic migration.

Direct migration implies that an entire resource is moved from hardware to software. Such migration relieves transistor count which can be "invested" into resources that have been proven useful in "fighting" with large off–chip delays. Examples of direct migration can be found in RISC processors for silicon technology. For example, in the Stanford University MIPS machine [15], the pipeline interlock mechanism was migrated from hardware to software [16]. In the Berkeley RISC machine, and several other machines, the delayed branch control mechanism was migrated from hardware to software [17].

Correct utilization of the direct migration principle can increase the efficiency of GaAs systems. However, examples of direct migration seem to be very difficult to find. For that reason, our attention has turned to catalytic migration. Catalytic migration is potentially less useful because (on average) the transistor count released through catalytic migration is less than the transistor count released through direct migration. However, examples of catalytic migrtion are much easier to find. In the case of catalytic migration, a relatively small piece of hardware (catalyst) is added to the processor, which enables a much larger piece of hardware (migrant) to be migrated from hardware to software. Before a catalyst was added, it was not possible (or did not pay) to move the migrant from hardware to software.

The whole concept can be placed into a more general context. First, as indicated above, the role of the catalyst can be to enable the migration which is otherwise not possible. This type of migration is referred to as *inductory catalytic migration.* Second, migration may be possible even before the catalyst was added, but the addition of the catalyst increases the efficiency of migration. This type of migration is refered to as *acceleratory catalytic migration.* In principle, there are two subcases of *acceleratory catalytic migration.* In the first subcase, the term efficiency refers to the speed

[5]"Fighting" the large off–chip delays is the major reason for turning to software (compile–time optimizations), but not the only reason.

of compiled HLL code. In the second subcase, the term efficiency means that the optimizing compiler is less difficult to generate (but the speed of compiled HLL code may not necessarily improve). The first subcase is referred to as *execution–oriented*, and the second subcase is referred to as *compilation–oriented*.

Elements of the above strategy can be found in the past work of others [1, 15]. However, initial similarities are limited only to the issues which are of a secondary nature for the strategy presented here. For example, the paper [18] points to the synergistic effects that could be created through interaction of the architecture and the optimizing compiler. In the paper that follows [19], the authors introduce the concept of integration as an approach that is potentially useful for GaAs microprocessors. Also, in some optimizing compilation related work, it was indicated that the code optimization problem could be more easily solved if some kind of hardware "help" is available [20, 21].

On the other hand, some of the solutions employed in modern microprocessors [22] can be treated as examples of catalytic migration, in a very wide sense. The authors of these solutions did not create them by trying to follow the guidelines of the strategy presented here. They simply got an idea that improves the efficiency of their microprocessor, and they implemented it. Obviously, catalytic migration can be useful in the silicon environment as well, but it yields the best results in the environment of GaAs, or similar technologies.

5. Examples of catalytic migration

Three examples of catalytic migration will be briefly described here. Other examples can be developed, too. However, the intention of this paper is not to provide an exhaustive list of catalytic migration examples, just to illustrate the concept. Further catalytic migration examples are the subject of a follow–up paper.

5.1 Catalitic migration example #1

The GE (RCA) GaAs RISC machine [9] uses the so called "hand–walking" solution to solve the problem of "scrambled load," typical of the multi–distance main memory (which can not be avoided in GaAs computer systems). The essence of the problem, and the "hand–walking" solution, is as follows (see Figure 1). In silicon, the worst case access time determines the value of the memory cycle. In GaAs, due to a relatively large ratio of off–chip to on-chip communication delays, this strategy makes no sense.

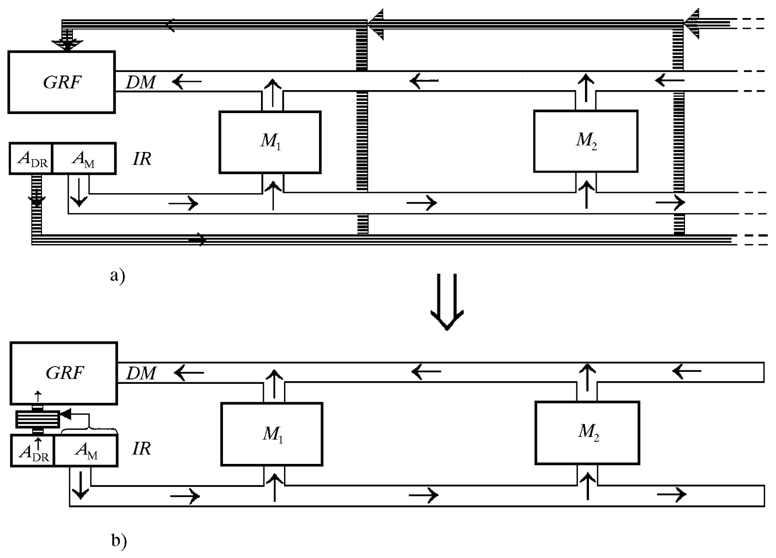

Figure 1. Example of an Inductory Catalytic Migration:
(a) Before migration (migrant shaded);
(b) after migration (catalyst shaded).
IR: Instruction register.
A_{DR}*: Address of the destination register for LOAD instructions.*
A_M*: Memory address for LOAD instructions.*
M_i*: Multidistance memory at distance* i $(i = 1, 2, \dots)$.
D_M*: Data from memory, for LOAD instructions.*
GRF: General register file.
DU: Delay unit which determines how long to keep A_{DR} *away from the general register file.*

Consequently, systems are designed with a multi–distance main memory, i.e. different parts of main memory have different access times. One of the problems with this is that a load from a distant memory, immediately followed by a load from a non–distant memory, may result in swapping of data between two registers, if a traditional register load design is used (destination register address sent directly from the instruction register to the address input of the register file). Actually, the later load from a non–distant memory may bring in the data item (from memory) sooner than the former load from a distant memory. With the above "traditional register load design," data would end up in wrong registers. One way to avoid this problem is by including one more bus in the system, a bus that would propagate the above destination register address to and from the main memory. This enables the destination register address to propagate to the main memory together with the load address, and back together with the load data. Consequently, data related to two successive loads can not be swapped. This technique is referred to as "hand–walking." In principle, the "hand–walking" bus can be migrated into software, in which case the optimizing compiler would "shuffle" the instructions around the load instructions, in order to move the load instructions far enough apart. When this is not possible, NOOPs would be inserted [16]. Unfortunately, in a typical programming environment, there is not enough static information available at compile time. The solution which goes along the guidelines of the catalytic migration approach implies the incorporation of a "small" piece of hardware (catalyst, shaded in Figure 1b), which would make sure that the rest of the relevant information becomes available at run time. In other words, the added piece of hardware, controlled by the code in execution, would enable the destination register address to stay away from the register file address input for the time which is exactly as long as needed (i.e., to be delayed as long as needed). Consequendy, the destination register propagation bus will be eliminated (migrant shaded in Figure 1a). This solution can be treated (conditionally[6]) as an example of *inductory catalytic migration*.

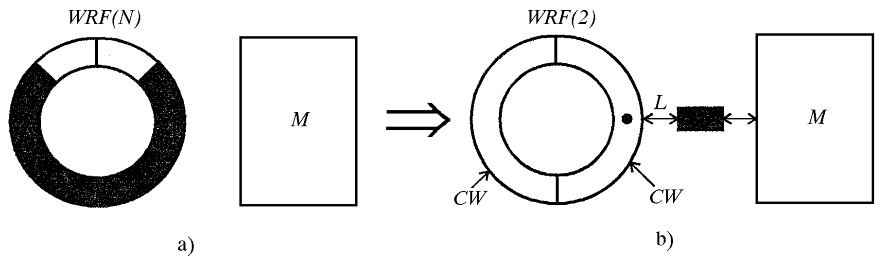

Figure 2. Example at an Execution–Oriented Acceleratory Catalytic Migration:
(a) Before migration (migrant shaded);
(b) After migration (catalyst shaded).
$WRF(N)$: Windowed register file (with or without overlapping) with N windows.
M: Memory
$WRF(2)$: Reduced windowed register file (with or without overlapping) with 2 windows.
CW: Current window
BW: Back–up window
DMA: Direct memory–access
L: Number of bits transferred by DMA in each DMA cycle ($L = 1, 2, \ldots$)

[6]This example contains also elements of other catalytic migration approaches.

5.2 Catalytic migration example #2

The Berkeley RISC machine contains an overlapping windowed register file, with N windows inside the CPU (see Figure 2). Even without the incorporation of a catalyst, a subset of these windows could be migrated from hardware to software. This can be done simply by letting the software maintain a subset of windows in the main memory. One way to make this migration more efficient in terms of the speed for compiled HLL code is to incorporate a "small" catalyst in the form of a bit–serial DMA channel (shaded in Figure 2b). In this way, up to $N-2$ overlapping windows (migrant, shaded in Figure 2a) can efficiently be migrated away from the CPU. Of the remaining two windows, one would serve as the "current" window, and the other one would serve as the "back–up" window. While the execution of the "current" procedure runs from the "current" window, the local variables (and other relevant parameters) of the "previous" procedure would be saved into the main memory through an L–bit–serial ($L = l, 2, \dots$) DMA transfer, and vice versa. The L–bit–serial DMA transfer is fast enough for these purposes, since the number of local variables (and other relevant parameters) is typically relatively small, and the DMA transfer time is very likely to take less time than the time to execute the "current" procedure from the "current" window. Data integrity is ensured via appropriate hardware. This is (conditionally[7]) an example of *execution–oriented acceleratory catalytic migration.*

5.3 Catalytic migration example #3

The GE (RCA) GaAs RISC machine [9], as well as some other GaAs machines which are now in the works, do have a relatively deep memory pipeline. If there exists no hardware interlock for delayed branching, the optimizing compiler will have to fill in the branch delay slots with other useful instructions, plus NOOPs, if the depth of the delayed branch slot is larger than the number of useful actions that can be found. The deeper the pipeline, the larger will be the percentage of NOOPs in the compiled HLL code. One way to ease this problem is to employ global rather than local code optimization techniques. However, this approach would make the optimizing compiler much more difficult to implement. Another way, which would make the optimizing compiler more efficient, without making it more difficult to implement, is deseribed here briefly. If two independent modules can be found, each can serve as a source of branch fill–in instructions for the other one, which would (approximately) double the number of useful instructions

[7]This example contains also elements of other catalytic migration approaches.

that a "local" code optimizer would be able to find[8]. However, for this to happen, a "small" catalyst would have to be added, in the form of some registers and control mechanisms duplicated (e.g., each of the two modules would have to have its own "minimal" register file acting like a catalyst). In conclusion, this solution makes it easier to build an optimizing compiler for a given level of branch delay fill–in success ratio. The approach could be extended to n independent modules. This solution represents (conditionally) an example of *compilation–oriented acceleratory catalytic migration.*

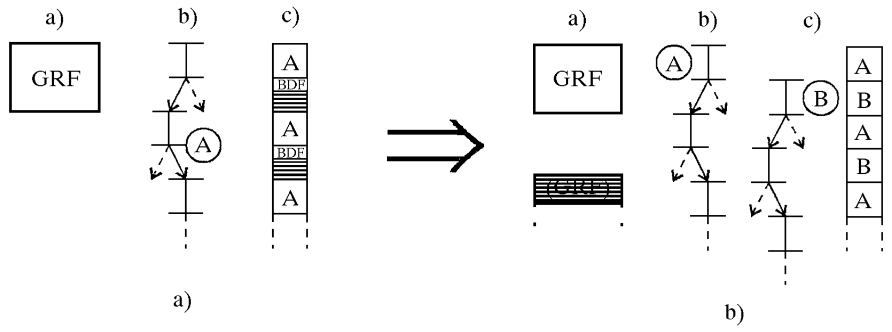

Figure 3. Example of a Compilation–Oriented Acceleratory Catalytic Migration:
(a) Before migration (migrant shaded).
aa) General register file (GRF).
ab) Execution trace of module A.
ac) Code of module A: BDF: Branch delay fill–in;
X: Unfilled part of delayed branch.
(b) After migration (catalyst shaded).
ba) Two general register files, one for module A, the other for module B.
S(GRF) is a subset of general register file.
bb) Execution traces for modules A and B.
bc) Code of module A, with sections of module B used to fill
in the delayed branch slots in module A.

6. Performance evaluation

Performance evaluation is a critical issue for any new concept. This is true for catalytic migration, as well. Not every example of catalytic migration is useful. Unfortunately, it can be extremely difficult to determine precisely if an example of catalytic migration really speeds up the execution

[8]Eliminated NOOPs (indicated with $\times$ in Figure 4a) can be conditionally treated as a migrant.

of the compiled HLL code (or whatever else is the criterion of efficiency), and for how much. Fortunately, functional description languages (like ISP' or VHDL), and good standard–cell placement and routing programs (like MP2D or TANNER) can be very useful.

In this paper, first the basic performance and cost/performance evaluation guidelines are given. After that, the results are shown for the application of the same guidelines to the three examples mentioned above.

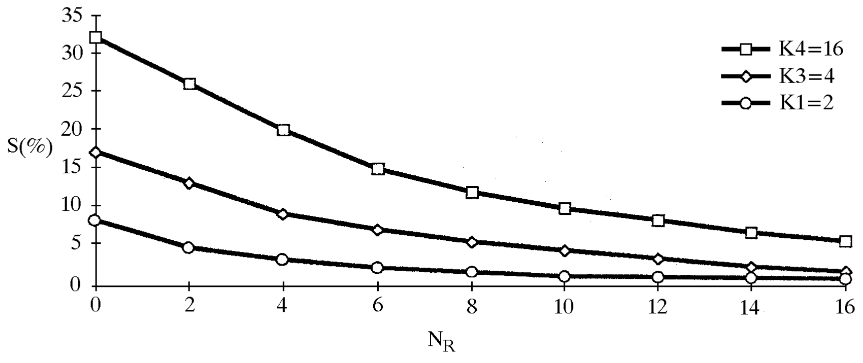

Figure 4. *Relative speed–up of the compiled code, assuming that the area saved by catalytic migration was used to increase the number of registers.*
S Code speed–up
N_R Initial number of registers, before migration
K_i $(i = 1, 2, 3)$: Additional number of registers due to area savings because of the utilization of catalytic migration.
This figure assumes that the ratio of of–chip to on–chip delays is as in [HelMil89], which is one instruction after one cycle (for cache hit), and one instruction after three cycles (for cache miss).

6.1 General discussion

The first issue is to determine the amount of VLSI area (or transistor count) which is released by catalytic migration under consideration. Obviously, the released VLSI area is equal to the difference of the area for the migrant and the area for the catalyst. Consequently, it is important to determine the area of the migrant and the area of the catalyst. An engineering approach would be to come up with an RTL level design of both resources, and then to estimate the layout area, using analytic, simulation or implementation tools. This approach is design–dependent, as well as technology dependent. Consequently, methods have to be explored that enable the determination of some kind of lower and upper bounds. The design and technology related choices (silicon vs GaAs, custom vs standard–cell, etc.) can be treated as parameters of the analysis.

The second issue is to determine the amount of the on–chip "delay fighting" resource that could be created on the above mentioned saved VLSI area. One possible simplification is to assume that the saved VLSI area would be invested into adding more registers, assuming that, before the migration the number of CPU registers is equal to N_R ($N_R = 0, 1, 2, \ldots$). Of course, the lower the initial number of registers, the larger will be the effect of catalytic migration. Consequently, a dependency function can be created. At one point, the function will have a "cut–off point" beyond which this particular type of "investment" does not make much sense any more. Of course, the "cut–off point" is dependent on the quality of the optimizing compiler, which makes the problem fairly complex and difficult for a "precise treatment." This opens up the next issue of importance, which is the hardware–to–software migration algorithm to be employed at compile time.

The third issue is the choice of the code optimization algorithm to be used for migration purposes. One approach is to work with the existing compilation technology [e.g., 16], in which case the performance evaluation results could be treated as the lower bound. The other approach is to introduce novel algorithms.

The final issue is benchmarking (or tracing). In other words, a representative set of benchmarks (or traces) has to be run through the system, both before and after the migration, in order to determine the effects of the migration, for various values of parameter N_R. What makes this approach difficult is the selection of representative benchmarks (traces), for the given application field.

Once a set of curves (dependency functions) is generated (speed–up versus initial number of registers), for different values of the chosen parameters, designers of future machines should be able to make appropriate decisions.

6.2 Performance estimation

A simulation experiment based on the ENDOT package was created to demonstrate the potential advantages of the three migration examples, for the case of the GE (RCA) 32–bit 200 MHz RISC machine [9]. For each example, a solution was assumed which releases certain transistor count that becomes available for the implementation of additional registers. The goal was to determine the performance improvement related only to the register count increase. The number of registers before migration was assumed to be N_R ($N_R = 1, 2, \dots$), and the register count increase was found to be K_1, K_2, and K_3 for the first, second, and third catalytic migration example, respectively.

Initial ENDOT simulator of the GaAs machine [9] was modified in order to reflect the above modifications. After that, a large predominatly numeric test program (which closely corresponds to a sponsor application) was executed on the ENDOT simulator, and relevant simulation data were collected. These data are given in Figure 4, for one specific set of values for K_1, K_2 and K_3. Values of K_1, K_2 and K_3 are design dependent, and their inter–relationship can vary with the design. For the design used in the specific experiment of Figure 4, it was $K_1 < K_3 < K_2$. Note that the value of K_3 is also dependent on the program characteristics and the optimizing compiler characteristics.

The curves in Figure 4 demonstrate that the effects of catalytic migration (related to designer's ability to put more registers on the chip) are larger for technologies with the limited transistor count on the chip, since (in that case) the initial number of registers (N_R) is smaller. The curves in Figure 4 also demonstrate that positive effects of catalytic migration can be fairly large.

7. Conclusion

This paper explains the rationales behind the concept of catalytic migration, it briefly discusses existing types of catalytic migration (through representative examples), and sheds some light on the related performance evaluation.

The role and importance of catalytic migration should not be misunderstood. It is applicable to both silicon and GaAs; however, the smaller the chip size and the larger the ratio of off–chip to on–chip delay, the better is the potential performance improvement due to catalytic migration. In other words, it is potentially advantageous not only for GaAs, but for any other technology which has large off–chip to on–chip delay ratio.

Also, catalytic migration is not to substitute for other possible approaches, but to complement them. For example, if the level of integration of GaAs chips reaches some fairly large number (e.g., one million transistors on a single chip), catalytic migration will not become obsolete at that time. The problem here is how to achieve the speed–up which is made possible by the technology. Putting more cache memory on the chip will help improve the absolute speed, but the technology–related speed–up could be approached only if the principles of catalytic migration are applied first.

In conclusion, one of the challenging research topics now is the creation of new and better examples of catalytic migration, as well as the methods for a reliable cost/performance evaluation, prior to implementation.

Ackowledgements

The author is thankful to S. Lakhani and P. Chow for the help related to the ENDOT simulation package.

REFERENCES

1. Patt Y.N.: *Real Machines: Design Choices and Engineering Trade–Offs.* IEEE Computer, January 1989, pp. 8–10.
2. Karp S., Roosild S.: *DARPA, SDI, and GaAs.* IEEE Computer, October 1986, pp. 17–19.
3. Gilbert B.K., Naused B.A., Schwab D.J., Thompson R.L.: *Signal Processors Based Upon GaAs ICs: The Need for a Wholistic Design Approach.* IEEE Computer, October 1986, pp. 29–44.
4. Milutinović V., Fura D., Editor: *Tutorial on GaAs Computer Design.* IEEE Computer Society Press, 1989.
5. Milutinović V., Editor: *Microprocessor Design for GaAs Technology.* Prentice–Hall, 1990.
6. Mudge T.: *GaAs Microprocessor Design at the University of Michigan.* Internal Report, Ann Arbor, Michigan, November 1988.
7. Fouts D.J., Johnson J.M., Butner S.E., Long S.I.: *System Architecture of a Gallium Arsenide One–Gigahertz Digital IC Tester.* IEEE Computer, My 1987.
8. McDonald J.F., Greub H.J., Steinvorth R.H., Donlan, B.J., Bergendahl A.S.: *Wafer Scale Interconnections for GaAs Packaging: Applications to RISC Architecture.* IEEE Computer, April 1987, pp. 58–70.
9. Helbig W., Milutinović V.: *A DCFL E/D MESFET GaAs Experimental RISC Machine.* IEEE Transactions on Computers, February 1989, pp. 263–274.
10. Milutinović V., Fura D., Helbig W.: *An Introduction to GaAs Microprocessor Architecture for VLSI.* IEEE Computer, March 1986, pp. 30–42.
11. Hwang K.: *Computer Arithmetic.* Wiley, 1979.
12. Milutinović V., Bettinger M., Helbig W.: *Adder Design Analysis for a 32–Bit GaAs Microprocessor.* IEE Proceedings, Part E, 1989.
13. Rasset T.L., Niederland R.A., Lane J.H., Geideman W.A.: *A 32-bit RISC Implemented in Enhancement–Mode JFET GaAs.* IEEE Computer, October 1986, pp. 60–68.
14. Fox E.R., Kiefer K.J., Vangen R.F., Whelen S.P.: *Reduced Instruction Set Architecture for a GaAs Microprocessor System.* IEEE Computer, October 1986, pp. 71–81.
15. Hennessy J.: *The MIPS Machine.* Digest of Papers, Spring COMPCON 82, San Francisco, February 1982, pp. 2–7.

16. Gross T.: *Code Optimization of Pipeline Constraints.* Technical Report, No. 83–225, Stanford University, December 1983.

17. Katevenis M.G.H.: *Reduced Instruction Set Computer Architectures for VLSI.* Technical Report, No. UCB/CSD 83/141 , University of Califomia at Berkeley, October 1983.

18. Milutinović V., Fura D., Helbig W., Linn J.: *Architecture/Compiler Synergisms in GaAs Computer Systems.* IEEE Computer, May 1987, pp. 72–93.

19. Dietz H., Chi C.H.: *A Compiler–Writer's View of GaAs Computer System Design.* Proceedings oj the 21-st Annual Hawaii International Conference on System Sciences, Kona, Hawaii, January 1988, pp. 256–265.

20. Katz R.: *Implementation of VLSl Systems.* Technical Report, No. UCB/CSD 86/259, University of California at Berkeley, September 1985.

21. Chow F.C.: *A Portable Machine–Independent Global Optimizer: Design and Measurements.* Technical Report, No. 83–254, Stanford University, December 1983.

22. Gimarc C., Milutinović V.: *A Survey of RISC Processors and Computers of the Mid–1980s.* IEEE Computer, September 1987, pp. 59–69.