VERSATILE PROCESSOR DESIGN FOR EFFICIENCY AND HIGH PERFORMANCE

Sotirios G. Ziavras

Department of Electrical and Computer Engineering

New Jersey Institute of Technology

Newark, New Jersey 07102

Email: ziavras@njit.edu, *Telephone*: (973) 596-5651, *Fax*: (973) 596-5680

Abstract. We present new architectural concepts for uniprocessor designs that conform to the data-driven computation paradigm. Usage of our *D*^{2}*-CPU* (Data-Driven processor) follows the natural flow of programs, minimizes the number of redundant operations, lowers the hardware cost, and reduces the power consumption. Instead of giving the CPU the privileged right of deciding what instructions to fetch in each cycle, instructions are entering the CPU when they are ready to execute or when all their operand(s) are to be available within a few clock cycles. Thus, the application-knowledgeable algorithm, rather than the application-ignorant CPU, is in control. It results in outstanding performance and elimination of large numbers of redundant operations that plague current processor designs. The latter, conventional CPUs are characterized by numerous redundant operations, such as the first memory cycle in instruction fetching that is part of any instruction cycle, and instruction and data prefetchings for instructions that are not always needed. A comparative analysis of our design with conventional designs proves that it is capable of better performance, simpler programming, and high efficiency.

Keywords -- Processor design, data-driven parallel computation model, comparative analysis.

1. INTRODUCTION

PC(Program Counter)-driven CPUs are very widely used. They are the result of more than 25 years of evolution. Performance improvements have primarily been the direct or indirect result of higher transistor densities and larger chip sizes. CPU designers have rarely attempted to circumvent this evolutionary path. However, general-purpose processors exhibit 100 to 1,000 times worse energy-delay product than ASICs. Now is the right time to carry out a sincere evaluation of current CPU design practices and start an earnest investigation of alternative design philosophies, before we reach the silicon technology's limits around the year 2010. According to Hennessy “The performance increases over the past 15 years have been truly amazing, but it will be hard to continue these trends by sticking to the basically evolutionary path we are on … Occasionally, we have processors that get so big they roll backwards down the hill, rolling over the design team. So we can keep on pushing our ever larger rocks up these hills, but it’s going to get harder and harder, and it may not be the smartest use of all our design

talent" [13]. Our research here stems from these and other observations. For example, current CPU designs are characterized by numerous redundant operations, do not match well with the natural execution of programs, have unreasonably high complexity, and consume significant power. We justify these observations in this section, by investigating current CPU design practices. We also summarize work related to the data-driven computation paradigm that underlies our design.

Current microprocessors implement wide instruction issue, out-of-order instruction execution, aggressive speculation, and in-order retirement of instructions [5]. They generally implement a small, dynamically changing window of dataflow execution. Under the pure *data-driven (dataflow) computation paradigm*, an instruction is executed as soon as its operands become available [3]; it carries with it its operand fields. An instruction's result is copied as soon as it is produced into all other instructions that need it as input, using different tokens. The data-driven model matches well with the natural flow of programs. It is not known how to implement this paradigm efficiently with current hardware design practices and silicon technologies. For example, the dataflow instruction firing rule can be implemented through state-dependent instruction completion [4]. Every memory reference is treated as multiple instructions. Evaluating simple arithmetic expressions is rather slow because it requires two operand fetch operations from the *activation frame* in the memory; this frame is allocated to each instruction just before it starts executing, to facilitate the storage of *token*s. A token includes a value, a pointer to the instruction that must be executed, and a pointer to an activation frame. The instruction contains an opcode, the activation frame offset where the token address match will occur, and the addresses of one or more instructions that need the result of this instruction. This dataflow implementation is characterized by significant computation and storage overheads.

The data-driven paradigm has been primarily employed in the implementation of parallel computers; it is applied to instructions running on different PC-driven processors. The majority of the dataflow multiprocessor designs include many compromises because they employ COTS processors [4, 17]. A data-driven processor was introduced in [18]; it utilizes a self-timed pipeline scheme to achieve distributed control. The data-driven paradigm can accommodate very long pipelines that are controlled independently, because packets flowing through them always contain enough information and data on the operations to be applied. However, the latter design also suffers from several constraints imposed by current design practices. Finally, several ASICs can be developed in VLSI when the dataflow graphs of specific application algorithms are given.

For *multithreaded* processors, instructions within a thread are issued according to the PC-driven model of computation. Instructions between threads are run based on data availability [10, 11, 12]. A large degree of thread-level parallelism is derived through a combination of programmer, compiler, and hardware efforts. COTS processors can implement non-preemptive multithreading, where a thread is left to run until completion. The compiler must guarantee that all data is available to the thread before it is activated. It must identify instructions that can be implemented with *split-phase operations.* For example, a load from remote memory is initiated within the thread where the load instruction appears. The instruction that

requires the returned value as input then resides in a different thread. EARTH is a multiprocessor with multithreaded nodes [14]. Each node contains a COTS RISC processor for executing threads sequentially and an ASIC synchronization unit that supports dataflow-like thread synchronizations, scheduling, and remote memory requests. Multithreading is not the most efficient solution because it does not implement the dataflow model at the instruction level for the entire program. Also, multithreading and prefetching significantly increase the memory bandwidth requirements [15]. The *Processing-In-Memory (PIM)* technique [8] integrates logic into memory chips to potentially reduce the required memory bandwidth.

Configurable processors have become a major research area, primarily due to the recent success of FPGAs [1, 7]; however, relevant techniques are not yet ripe for general-purpose computing. Dynamic *multiple-issue processors* apply superscaling with multiple copies of commonly used functional units. VLIW processors are static multiple-issue machines without flexibility at run time [16]. The Intel IA-64 is the first commercially available microprocessor based on the *EPIC (Explicit Parallel Instruction Computing)* design philosophy [9]. Contrary to superscalar processors that yield a high degree of ILP at the expense of increased hardware complexity, the EPIC approach retains VLIW's philosophy of statically generating the execution schedule, while improving the processor's capabilities to better cope with dynamic factors. Finally, the high complexity of individual processors has a dramatically negative effect on the overall complexity and performance of parallel computers [2, 6].

To summarize, a close look at the current RISC, CISC, and VLIW (i.e., prevalent PC-driven) processor designs shows several deficiencies. They are characterized by large amounts of redundant operations and low utilization of *productive resources* (i.e., directly related to the implementation of application algorithms). The first phase of the instruction fetch operation is required only because of the chosen computation model (i.e., PC-driven); that is, this CPU request to the memory is not part of any application algorithm but is the result of centralized control during program execution. To alleviate the corresponding time overhead, current implementations use instruction prefetching with an instruction cache; many silicon transistors are then wasted that could, otherwise, be used in more productive tasks. Also, the operands do not often follow their instructions to the CPU; the only exceptions are with instructions that either use immediate data or their operands reside in CPU registers. Additional fetch cycles may then be needed to fetch these operands from either the main memory or the attached cache. However, these fetch cycles also should be avoided, if possible. They are unavoidable even with dataflow designs that use activation frames. Again, to mitigate this problem current designs choose data cache memories; the corresponding transistors could, otherwise, be used in more productive tasks. In contrast, in pure dataflow computing the instructions go to the execution unit(s) on their own, along with their operand(s), as soon as they are ready to execute.

Our D^2-CPU processor design philosophy is based on the pure data-driven computation paradigm and, like RISC and VLIW designs, promotes a small set of rather simple instructions, for ease and efficiency of instruction decoding and implementation. Also, register files within the processing unit contain each time instructions that will be definitely executed soon. Such a direct

hardware support is needed if our ultimate objective is to eventually fully integrate this computation paradigm into mainstream computing. The data-driven execution model is applied simultaneously to all instructions in the program. Section 2 introduces new design concepts that lead to D^2-CPU implementations for higher performance, ease of programmability, and low power consumption. Sections 3 and 4 present the details of our D^2-CPU and a comparative analysis with conventional designs.

2. INTRODUCTION OF THE D^2-CPU

Some definitions are pertinent to the data-driven computation paradigm. *Instruction firing* is the departure of an instruction for the execution unit just after all of its operands become available. *Token propagation* is the propagation of an instruction's result to other instructions that need it, as soon as the instruction completes execution. *Instruction dissolvement* is the destruction of an instruction just after it produces all of its tokens. The main *advantages* of data-driven computation are:

- Instructions are executed according to the natural flow of data propagated in the program.
- Most often, there is a high degree of embedded parallelism in programs and, therefore, very high performance is possible.
- It is free of any side effects (because of the natural flow of data that guides the execution of instructions).
- It reduces the effect of memory-access latency because all operands are attached to their instructions.
- It naturally supports very long, distributed, and autonomous superpipelining.
- Clock skewing is not an issue and, therefore, there is no need to synchronize all functional units.

The main *disadvantages* of data-driven computation are:

- Increased communication (or memory access) overhead because of explicit token passing.
- Instructions are forced to use their data (in incoming tokens) when they arrive, even if they are not needed at that time.
- The manipulation of large data structures becomes cumbersome because of the token-passing approach.
- The hardware for matching recipient instruction addresses in the memory with tokens may be complex and expensive.

Advances in current CPU designs lack the potential for dramatic performance improvements because they do not match well with the natural execution of programs. To alleviate this critical problem designers often apply expensive techniques. However, the relevant hardware is not used to directly run part of the original program. Instead, it is used in “unproductive” tasks that result in small utilization of the productive system resources. For example, the memory latency is reduced with expensive means, such as preprocessing, instruction/data prefetching, cache, internal data forwarding, etc. The drawbacks are:

- We waste numerous, otherwise precious, on-chip resources. Many hundreds of thousands or millions of transistors are needed to implement some of the above techniques within a single CPU.

- Power consumption increases for two reasons. Firstly, the overhead of the instruction fetch cycle appears for each instruction in the program. These interchip data transfers are quite expensive. Secondly, unnecessary power consumption results from prefetching unneeded instructions and data into caches. Mobile computing systems need to be power efficient.
- Numerous cycles are wasted with hardware exceptions. After the CPU gets informed about the external event, it has to store the current state of the machine and then fetch code to run the corresponding interrupt service routine. If the appropriate context switching is instead selected outside of the processing unit, then the appropriate instructions can arrive promptly.

The major requirements for single CPU designs that could avoid old pitfalls of the type discussed above are:

- Programs are developed easily using a fine-grain graphical (or equivalent) language that shows explicitly all data dependencies among the instructions. Thus, no application inherent parallelism is hidden in the program.
- Instructions contain all their operand fields.
- A software preprocessor finds all instructions in the program that can run in the beginning because of non-existent data dependencies. These head instructions are to be sent first to the execution unit.
- Following the head instructions to the execution unit are instructions that are to receive all their (input) operands from one or more head instructions. They proceed for execution just after they receive their operand(s). In general, all instructions follow to the execution unit the instructions that are to produce values for all of their missing operands.
- Instructions that are to receive one or more operands from instructions that are ready to execute, but they will still be missing one or more operands leave for an external cache, namely the EXT-CACHE, where they wait to receive their tokens. To reduce the traffic, instructions that will receive the same result are grouped together in the cache in an effort to collectively receive a single token that can be used to write all relevant operand fields. If not all of the token receiving instructions can fit in the EXT-CACHE, then a linked list is created in the memory for instructions that do not fit.
- Only one copy of each instruction, including its operand(s), resides at any given time within the entire machine. This is in contrast to the wide redundancy of instructions and data present in the cache, memory, and CPU of conventional machines.
- Instructions do not keep pointers to their parent instructions. They sleep until they are forced into the EXT-CACHE or the execution unit, in order to receive their token(s). This lack of backward pointers does not seem to permit instruction relocation which is needed to implement multiprogramming and/or virtual memory. Without its parent pointer(s), a relocated instruction cannot inform its parent(s) about its new location, for appropriate token propagation. Nevertheless, we have devised a technique that permits token forwarding inexpensively for relocated instructions.
- After an instruction is executed, it is dissolved. However, special care is needed for instructions that have to be reused in software loops. A relevant technique that permits instruction reuse is presented in the next section.

- Instructions have unique IDs for token passing only while they reside outside of the execution unit. These IDs are used to find instructions and force them into the EXT-CACHE or the execution unit. In the latter case, an interface actually keeps track of these IDs, so that minimal information is manipulated or stored in precious execution unit resources.

3. D^2-CPU DESIGN

The requirements listed above are taken into account to derive our CPU design. We employ advances in PIM, cache-memory, and IC technologies to implement very efficiently the data-driven computation paradigm. The proposed, namely *D^2-CPU* (where D^2 stands for Data-Driven), design is shown in Figure 1. The EXT-CACHE (EXTernal to the execution unit CACHE) is distributed; it is a collection of $DSRAM_i$s (Distributed SRAMs), for $i = 0, 1, 2, \ldots, 2^d-1$, one DSRAM module per DRAM module. The DRAMs originally contain the program code.

Each instruction comprises an opcode field and, without loss of generality, up to two operand fields. If the instruction is to produce a result, then a variable-length sequential list follows. This list contains pairs of recipient instruction IDs and single-bit operand locators, so that data produced by the former instruction can be propagated to the latter instructions; the operand locator bit is *0* or *1* for the first or second operand field, respectively. The instruction's context ID is appended to the actual instruction ID to form a new ID. When the instruction goes to the ERU for execution, these pairs go to their associated DSRAMs and wait there to receive their result tokens; the ID also of the result producing instruction is kept in each DSRAM to facilitate the identification of the incoming token for data replication.

The *Execution-Ready Unit (ERU)* comprises:

- The *Processing Unit (PU)* where the operations specified by the instructions are executed. It contains several functional units that can be used by a single, or simultaneously by multiple, instructions. Its design follows the RISC model.
- The *ERU-SRAM (static RAM in the ERU)* contains instructions ready to execute. However, these instructions cannot proceed to the PU yet because the functional units that they require are currently in use. When a functional unit becomes available, its ID is sent to this cache to find an instruction that requires this unit.
- The *SRAM** contains instructions with one or more unfilled operand fields that are all to be written by one or more instructions currently residing in the PU and/or ERU-SRAM. Therefore, the former instructions will execute in the very near future. For each instruction in the ERU-SRAM, there is a hardware count of its result recipient instructions in the SRAM*. When a functional unit becoming available broadcasts a signal to the ERU-SRAM, the relevant instruction with the largest number of recipient instructions in the SRAM* wins.

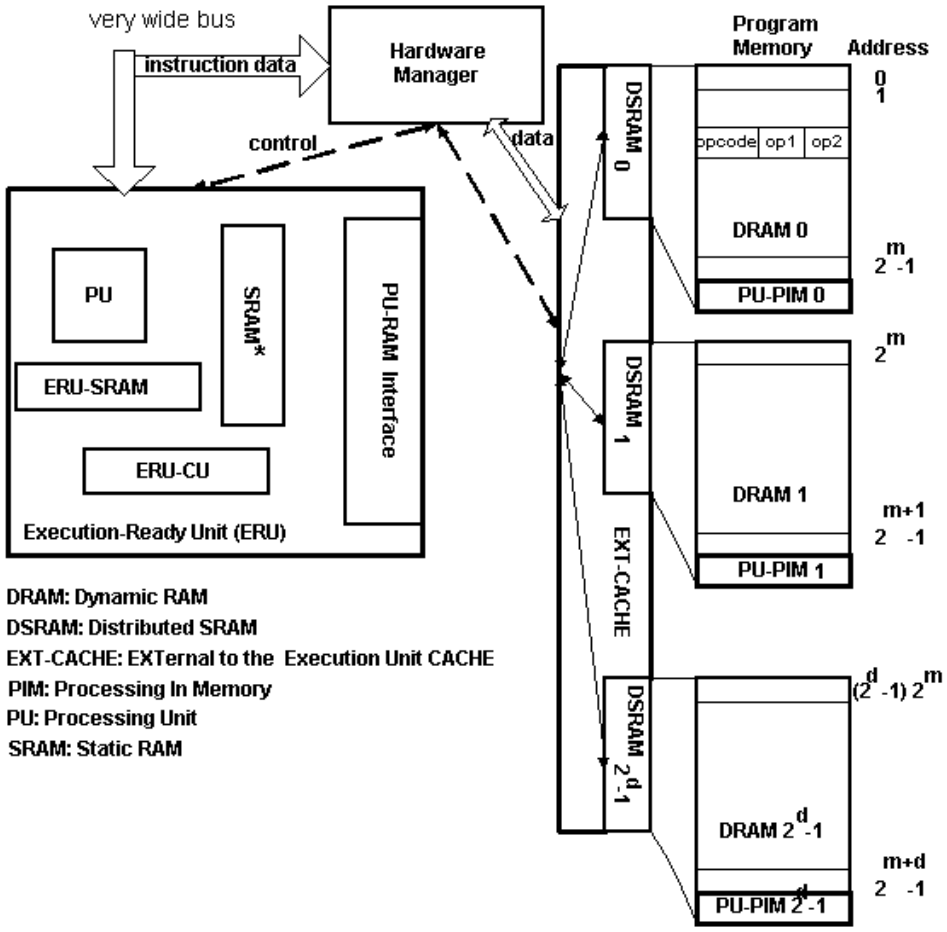

Figure 1. Architecture of the D^2-CPU and its interface.

- The *ERU-CU* (control unit) keeps track of the total number of result recipient instructions in the SRAM* for each instruction currently in the ERU-SRAM. It also facilitates data forwarding within the ERU, for the SRAM*.
- The *PU-RAM interface* receives from the hardware manager all instructions entering the ERU. It distributes them accordingly to the PU, ERU-SRAM, and SRAM*. When an instruction produces a result, the PU-RAM interface forwards to the hardware manager the *virtual ID* of this instruction followed by the result. The virtual ID of an instruction is a temporary ID assigned to it by the hardware manager just before it enters the ERU (discussed below).

The *hardware manager* performs the following tasks:

- It initially sends the head instructions of the program to the ERU for execution.
- Whenever one of the remaining instructions proceeds to the ERU on its own, it first makes a request to the hardware manager for a virtual ID. This ID will uniquely identify the instruction during its residency in the ERU. It is a small number in the range *0* to *n-1*, where *n* is the maximum number of instructions that can reside simultaneously in the ERU. These IDs are

recycled. The reasons for assigning new IDs to instructions (instead of keeping their long external/original IDs) are: (a) to minimize the required bandwidth between the ERU and external components; (b) to minimize the size of ERU-internal resources storing the instructions' IDs; and (3) to minimize the required resources that process ERU-resident information.

- It maintains a table that can be accessed to quickly translate on-the-fly short (virtual) IDs into long (original) instruction IDs for results broadcast by the PU-RAM interface in the ERU. The modified tokens are then forwarded to the EXT-CACHE where they are consumed by their recipient instructions, as described in the next paragraph.

The *EXT-CACHE* contains at any time up to three classes of instructions that are to receive tokens from the ERU:

- **Class A**. Instructions with two unfilled operand fields. One field is to receive data from an instruction in the ERU.
- **Class B**. Instructions with one unfilled operand field for which the token is to arrive from an instruction currently residing in the ERU. These instructions cannot fit in the ERU because the SRAM* is fully occupied.
- **Class C**. Instructions that are not missing any operands but they cannot fit in the ERU-SRAM because it is fully occupied.

The currently achievable transistor density for chips allows the implementation of large enough memories to realize the ERU-SRAM, SRAM*, and DSRAM components so that they very rarely overflow. Without hardware faults, there is no appearance of deadlocks. Even if the ERU-SRAM is fully occupied at some time, its instructions will execute in the near future because the PU will be released soon by the currently executing instructions. If one or more instructions outside of the ERU are ready to execute but cannot enter the ERU because the ERU-SRAM is fully occupied, then they wait in an external queue until space is released in the ERU-SRAM. A similar technique is applied if the SRAM* is fully occupied.

The virtual IDs assigned by the hardware manager to instructions departing for the ERU are q-bit numbers, where $q=log_2 n$ (n is the maximum number of instructions that can reside simultaneously in the ERU, and we assume that it is a power of *2*). For each *program memory module* $DRAM_i$, there is a distinct cache $DSRAM_i$ in the EXT-CACHE, for $i=0, 1, 2, ..., 2^d-1$, containing instructions that are to receive tokens from instructions currently residing in the ERU. Entries in this cache have the format shown in Figure 2, and the fields in each entry are:

- *IID*: the *token-sending instruction ID* assigned originally by the compiler-loader combination.
- *IAD*: the *token-receiving instruction ID*. It is the ID of an instruction that came from the corresponding $DRAM_i$. This instruction will consume a token produced by the instruction with ID stored in the *IID* field of this entry. More than one entries may have the same value in their *IID* field.
- *OPFL*: *operand field locator*. A value of zero (or one) in this one-bit field means that the data from the received token is for the first (or second) operand field of the instruction with ID *IAD*.
- INSTR: the *token receiving instruction*. It is the instruction with original ID stored in the IAD field of this entry.

Each token leaving the ERU with the result also contains the virtual ID of the instruction that produced the result. The hardware manager changes on the fly the virtual ID to the original ID of this sending instruction and broadcasts this token to all DSRAMs in the EXT-CACHE. We assume up to 2^s original instruction IDs and r-bit instructions.

Size of the field (bits): *s* *s* *1* *r*

Figure 2: The format of entries in DSRAMs.

The ERU receives instructions for execution from the hardware manager; the latter can also choose in the EXT-CACHE an active context ID. The hardware manager forces instructions into the ERU by first storing them into FIFO buffers and then prompting the ERU to read from these buffers using a very wide bus. Asynchronous communications with appropriate acknowledgments between these two units can achieve this task. Ideally, each DSRAM should interface the ERU with its own FIFO buffer and corresponding control signals, and the ERU should have separate buses to access separate FIFOs. The amount of ERU outward data transfers is minimal because produced data tokens are often forwarded internally to the local SRAM*. When the PU produces the result of an instruction, this result is forwarded to the ERU-CU and PU-RAM interface units. The former unit stores the result appropriately into recipient instructions, if any, in the SRAM*. The latter unit sends a special token to the hardware manager, so that the instructions outside of the ERU that are to be recipients of the produced data can receive the result. A single token is produced by the PU-RAM interface unit in order to minimize the amount of traffic. As emphasized earlier, the special token comprises two fields that contain the produced data and the ERU-resident ID of the instruction that produced the result, respectively. The hardware manager replaces this short ID with the instruction's original ID and forwards the token to all DSRAMs. Fetching short IDs is not a heavy penalty to pay for the elimination of the PC. Let us not forget that a PC-driven CPU requires the implementation of a wide address bus and appropriate control lines. Our ERU needs fewer pins to fetch (in one memory bus access) this ID, whereas PC-driven CPUs need more pins to access instructions (in two memory bus accesses, by first sending out a RAM address and then receiving the data).

For a fixed value of i, the tasks carried out by the *PU-PIM*$_i$ are:

- It loads the DSRAM$_i$ with all those instructions from the DRAM$_i$ that are to receive tokens from instructions leaving for the ERU and are also missing data for two operand fields. Also, it always updates appropriately the DSRAM$_i$ directory.
- It removes instructions from the DRAM$_i$ that are not to be executed further because of loop exiting.

- It maintains three distinct lists of addresses for instructions in the $DRAM_i$, if any, that do not fit in the EXT-CACHE, ERU-SRAM, and SRAM*, respectively.
- It copies data from tokens broadcast by the ERU (via the hardware manager) into the appropriate operand fields of instructions appearing in the aforementioned overflow lists for the EXT-CACHE and SRAM* units.
- It carries out garbage collection in the $DRAM_i$ (e.g., because of dissolved instructions).
- It finds instructions in the $DRAM_i$ and $DSRAM_i$ that are to receive their last operand from instructions leaving for the ERU and forwards them to the hardware manager that finally stores them into the SRAM*.
- It services requests by the program loader and the operating system for instruction loading and relocation in the $DRAM_i$. Justification of the need for instruction relocation and its solution are presented in the next subsection.

In order to speed up the overall process, the PU-PIM_i unit may be replicated many times for each module $DRAM_i$. The ensemble of all components that form the (PU-PIM_i, $DRAM_i$) pair is then looked upon as a parallel shared-memory machine.

3.1. Support for Instruction Relocation

If data-driven computation is to compete with, and surpass in success, the PC-driven paradigm, then it has to provide for *multiprogramming* and *virtual memory*. Both of them, in turn, require support for instruction relocation. Instruction relocation in data-driven computation seems to be a very difficult problem to solve because of the need for token passing with ever-changing instruction addresses. We have devised the following solution for instruction relocation in a way that token passing using original instruction IDs is still possible:

- The compiler-loader combination assign the original instruction IDs to correspond to absolute memory addresses. If a memory location is free at that time, then the corresponding instruction, if any, is loaded there. The instruction's context ID is also stored in the memory along with that instruction. If that memory location is occupied by another instruction, then the former instruction is relocated early according to the following method.
- We assume *instruction relocation* limited to a single program module $DRAM_i$. A distinct *ID memory* module ID_MEM_i, implemented in DRAM technology, of the same size is associated with each program memory module $DRAM_i$. The j^{th} entry in the ID_MEM_i contains the starting address of a hash table with pointers to all instructions with original ID equal to *j*, but with different context IDs, for $j = 0, 1, 2, ..., 2^m-1$. When an instruction with original ID F relocates in the $DRAM_i$, then the respective PU-PIM_i unit stores in the hash table pointed at by the value in address F of the ID_MEM_i the context ID and the new address of this instruction. The hashing technique applied to locate the new address of an instruction uses the program/context ID as the key.

- The PU-PIM$_i$ unit keeps track of the location of all instructions in the DRAM$_i$. More specifically, it updates the hash tables to point to the new location of instructions. Each instruction carries its original ID and context ID. Whenever it is moved to a new location, the corresponding pointer in the associated hash table is updated by the PU-PIM$_i$.
- If an instruction with context ID Q goes to the ERU for execution, then the instructions that are to receive its tokens must go to the SRAM* or EXT-CACHE. To find the current location of an instruction in the latter group, with original ID F and context ID Q, the PU-PIM$_i$ performs the following tasks:
  - If the instruction at location F of the DRAM$_i$ has ID F and context ID Q, then the instruction has been located.
  - Otherwise, the hash function with key Q is applied for the table pointed at by the entry at location F of the ID_MEM$_i$, in order to find the new location in the DRAM$_i$ of the token recipient instruction.

3.2. Handling Exceptions

We distinguish between software and hardware exceptions (interrupts). For example, a software exception will occur if an erroneous result is to be produced by an instruction (e.g., division by zero). An appropriate set of instructions are normally executed before the instruction that may produce the erroneous result. If their code determines that the erroneous result will show up with the execution of the latter instruction, then a thread of instructions (i.e., an exception routine) are activated (by passing an *activation token*) to deal with this problem. Among other tasks, this thread removes from the system the "faulty" instruction; although this instruction will never be executed because its full set of operands will never become available to it, it must still be removed to free memory/system space. Of course, some software exception routines must always be available at run time. Replication of the exception routine code an unknown number of times may then be needed. The hardware manager is instructed (by the instruction that makes the determination about the erroneous result occurrence) to use the PU-PIMs in replication of code stored in the DRAM memory. The hardware manager sets as context ID of the replicated code the context ID of the determining instruction. It is not necessary to halt the execution of instructions that do not belong to the exception routine. If required to do so, however, because of high priority, then the hardware manager can temporarily ignore all instructions in the EXT-CACHE with context ID different from that of the exception determining instruction.

Hardware exceptions can be dealt with in the following manner. The instructions for exception routines are initially stored in the DRAM memory. The hardware manager receives the exception request along with an exception ID. This ID is used to uniquely determine the address of the first instruction in the exception routine. The hardware manager forces the PU-PIMs to make a copy of the exception routine code and also sends an activation token to the first (i.e., head) instruction. The instruction is eventually fetched from a DRAM into the ERU via the hardware manager. From this point on, the hardware manager disables all transfers to the ERU of instructions that have different context ID than this exception ID. It also sends this

exception ID to the ERU to disable the execution of instructions with different context IDs. Every (software or hardware) exception routine contains a *last instruction* that upon execution forces (by passing a token) the hardware manager to enable all context IDs for the resumption of program execution. Our design can easily deal with different priorities for exceptions by keeping active, each time an exception occurs, all contexts with ID greater than or equal to the ID (priority) of the exception.

3.3. Program Loop Implementation

To facilitate efficient memory management, loop instructions should not be repeated unwisely in the memory. A bit in each instruction can indicate its inclusion in a loop, so that the instruction can be preserved at the end of its execution for future executions. Only its operand fields are emptied, if needed, after each execution. Upon exiting a loop, the last instruction sends a special *dissolve token* to the first instruction in a special routine that removes all loop instructions from the memory; only the PU-PIMs are involved in this process. It is not necessary to have a single loop exit. As for conditional branching, instructions that are not executed are dissolved similarly by special routines. However, we do not cover all details here because more work may be needed for a better solution with higher performance and lower cost.

3.4. Similarities and Differences Between the D^2-CPU and VLIW Architectures

Similarly to VLIW architectures, our ERU requires a wide instruction bus because each DSRAM memory can send one instruction (that also contains up to two operands) in a single memory cycle. Also, we assume asynchronous transfers between the ERU and individual DSRAMs, whereas the simultaneous transfer of all components (i.e., simple instructions) of a long instruction is a prerequisite for the execution of instructions on VLIW architectures. Therefore, the D^2-CPU design has these obvious advantages over VLIWs:

- Dramatic reduction in redundant operations (because VLIWs are PC-driven).
- It is faster because of asynchronous instruction and data transfers (when a DSRAM is ready, it just sends its instruction to the ERU without waiting for the other DSRAMs to also become ready).
- Any code is portable since there is no need for a compiler to group together simple instructions into large ones that are executable on the specific machine. In contrast, simple instructions are dispatched to the ERU.

4. COMPARATIVE ANALYSIS

Let us now carry out a comparative evaluation of our D^2-CPU with conventional PC-driven CPUs. Comparisons will be made under various criteria. For the sake of simplicity, we assume non-pipelined units.

4.1. Operation Redundancy

The main purpose for the introduction of the D^2-CPU design was to dramatically reduce the number of redundant operations during program execution and to use the corresponding resources for the efficient implementation of operations explicitly requested by the executing code. The first step was to completely eliminate the instruction fetch cycle, which is implemented for every instruction in the PC-driven model, by adopting the data-driven model of execution. Assume the following parameters:

- *k*: the number of clock cycles in that part of the memory fetch cycle that begins with the initiation of the cycle by the microprocessor and ends when the memory starts sending the data on the memory bus.
- *f*: the probability for a page fault to appear (that is, data still reside in secondary devices).
- *p*: the penalty in number of clock cycles resulting from a page fault.
- *N*: the total number of instructions in the program that have to be executed.

We can assume that our D^2-CPU has a zero probability for page faults because: (a) there can always be enough free memory space to load code since instructions are dissolved just after their execution and (b) pages are created based on the data dependence graphs of programs that show explicit data forwarding and therefore imply known snapshots for future instruction executions. The data-driven computation paradigm, and therefore our CPU, requires to write the result of each operation into instructions that will use it as an input operand. The recipient instructions will arrive later for execution, along with their operands. Under the PC-driven paradigm, the result is written once but it is fetched separately for individual instructions by implementing memory fetch cycles that use both directions of the memory bus in two consecutive phases.

Therefore, the redundancy in operations for the PC-driven design results in an overhead of $T_{operations} = O(N*(k+f*p))$. In fact, this number may be higher with pipelining because of unnecessary prefetch operations for failed branch predictions. It also increases further for cache memory.

4.2. Storage Redundancy

Contrary to PC-driven CPUs that attempt to hide memory latency by keeping duplicates of the "most probable to use in the near future" information in caches attached to the CPU, our design keeps only a single copy of each "yet to be executed" instruction in the system. Of course, the D^2-CPU does not need (cache to keep) duplicates of instructions because the execution path is known at any time from the data dependence graph. Also, instructions are inherently dissolved for our design just after they are executed, and therefore they free space, and simplify numerous compiler and operating system tasks. Therefore, the PC-driven design has storage overhead *O(N)* words because it does not dissolve instructions after they are executed and also keeps multiple copies of many instructions. Assuming a fixed number of operands per instruction, the storage redundancy

associated with the D^2-CPU due to having multiple copies of the operands (i.e., one copy per instruction) is *O(N)*. However, the PC-driven design requires to store the operand addresses for instructions; these addresses also occupy space *O(N)*. The redundancy associated with storing on the D^2-CPU the addresses of instructions that are to receive data tokens is also *O(N)*. Therefore, all these counts of redundancy cancel each other out in comparative analysis.

4.3. Turnaround Time

We will calculate the expected peak performance of each CPU. We make the following assumptions:

- *N*: the total number of instructions in the data-dependence graph of the program. The program contains only zero-, one-, and two-operand instructions.
- *N'*: the total number of zero-operand instructions when we start the execution of the program.
- *q*: the ratio of one-operand, uniformly distributed, instructions in the remaining code (of *N-N'* instructions).
- Every instruction produces a one-word result. The entire program is originally stored in the DRAM and there is no need for relocation of instructions. The program does not contain any software loops. Every operand fits in a single word. Every instruction opcode fits in a single word for the PC-driven CPU. Every instruction opcode and the corresponding instruction's ERU-resident ID are stored in a single word for the D^2-CPU. Memory addresses are one-word long.
- *t*: the fixed number of outgoing edges for each (including leaf) instruction in the data-dependence graph.
- *B*: the maximum bandwidth of the processor-DRAM bus in words per cycle.
- The hardware manager for the D^2-CPU replaces original instruction IDs with ERU-resident (virtual) IDs, and vice versa, on the fly without any overhead. The PU-PIMs do not consume any extra cycles for garbage collection.
- The order chosen for writing data from a produced token into the *t* recipient instructions does not affect the performance of the D^2-CPU. For any executed instruction, its token recipient instructions are uniformly distributed among the DRAMs.
- PU-PIMs write data from a produced token at the collective rate of *B* recipient instructions per cycle.
- *c*: the total capacity, expressed in number of words, of registers in the PC-driven CPU. It is the same as the total cumulative capacity of the ERU-SRAM and SRAM* units in the D^2-CPU. It is important to note, however, that the capacity of the latter CPU can be much larger because of its much lower complexity.
- *E*: the peak execution rate, in number of instructions per cycle, for ALU operations in both CPUs.
- Instructions in the program do not have any resource dependencies.
- The CPUs do not contain cache where instructions or data could be prefetched.
- The PU-PIMs fetch instructions into their DSRAMs at a rate that usage of the DRAMs is transparent to the ERU.

4.3.1. Turnaround Time on the PC-Driven CPU

The first component of the total turnaround time is the amount of time required to fetch all instruction opcodes into the CPU. The unidirectional effective bandwidth of the CPU-DRAM bus is *B/2* words per cycle, because it is divided equally between the address and data buses (each datum or address fits in a single word); also, two transfer phases of opposite direction are implemented for each instruction because the instruction's address must be first sent out using the memory address register (MAR) in the CPU. Thus, the total number of cycles needed to fetch all instruction opcodes is

$$T_{instr-fetch}^{PC} = \frac{N}{B/4} = \frac{4N}{B}.$$

The time to fetch all instruction operands is given as the summation of fetch times for one- and two-operand instructions

$$T_{oper-fetch-memory}^{PC} = \frac{4q(N-N')}{B} + \frac{8(1-q)(N-N')}{B} = 4(2-q)\frac{N-N'}{B}.$$

The total execution time for all instructions is given by

$$T_{execute}^{PC} = \frac{N}{E}.$$

The time required for instructions to store their result into the memory is

$$T_{res-store}^{PC} = \frac{4N}{B}.$$

We need to take into account that some instructions may find one or both of their operands in a CPU register; all instructions also store their result into the memory. The CPU space devoted to the implementation of registers in the PC-driven CPU is identical to the space used by the ERU-SRAM and SRAM* units in the D^2-CPU. It is *c* words. Therefore, the PC-driven CPU comprises *c* general-purpose registers (GPRs), where each register contains a single word. Assume that an instruction always stores its result into a GPR, and it is kept there for other instructions to use for the largest possible number of consecutive instruction cycles. Also, the instructions in the program uniformly utilize the registers. The expected maximum number of instruction cycles for which a result will occupy a given GPR is

$$a = \frac{cT_{execute}^{PC}}{N} = \frac{c}{E}$$

where the numerator represents the cumulative number of registers available for the entire program execution and the denominator represents the total number of results produced by instructions. This number of cycles corresponds to having $a/(2-q)$ complete instructions that use these registers at any time. We have to subtract from the total operand fetch time

$$T^{PC}_{oper-fetch-reduce} = \begin{cases} 4\dfrac{a}{2-q}\dfrac{N-N'}{B}, & if \quad t > \dfrac{a}{2-q} \\ T^{PC}_{oper-fetch-memory}, & if \quad t \le \dfrac{a}{2-q} \end{cases}.$$

4.3.2. Turnaround Time on the D^2-CPU

The first instructions to arrive at the ERU are the N' zero-operand instructions. This phase requires time equal to N'/B cycles because each such instruction occupies a single word and all transfers are unidirectional. These instructions then consume N'/E cycles to execute, and t copies are made of every instruction's result for token receiving instructions. The expected maximum number of instructions that can be transferred in any other single cycle from the hardware manager to the ERU is

$$\Delta = \frac{B}{3-q}$$

where the bandwidth B of the channel is divided by the expected average length of an instruction in words.

During the execution of the first N' instructions, $\varepsilon = DN'/E$ new instructions enter the ERU (let N' be a multiple of E). Since our ultimate objective here is to compare the performance of the two designs under the assumption of optimality for program runs, all of these e instructions will receive their tokens from the former instructions and will then be ready to execute. Assume that $B=c$ and $D=E$, so that units do not overflow. If the e new instructions are uniformly distributed, they collectively consume $(2-q)e$ of the produced tokens; this number is found from $qe+2(1-q)e$. Thus, the total number of cycles consumed in step 1 is

$$T^{D2}_1 = \frac{N'}{B} + \frac{N'}{E}.$$

In the next step, all N' results produced earlier are also transmitted outside of the ERU and consume N'/B cycles, and the latter e instructions are executed in the PU and consume e/E cycles. Therefore, the number of cycles consumed in this step is

$$T^{D2}_2 = \max\{\frac{N'}{B}, \frac{e}{E}\}.$$

Assume that $N'=E$, for the sake of simplicity. If $N>>N'$, this assumption has negligible effect on the result. In each remaining step, D new instructions enter the ERU and consume $(2-q)D$ tokens produced in the preceding step. All D results are also transmitted outside of the ERU. The total number of tokens produced in the program by all instructions will be Nt, while

(2-q)(N-N') will be consumed by instructions that utilize the ERU; the remaining tokens, we assume, are used to change the state of the machine and/or communicate with peripheral devices. In practice, an average value of *t* is between *1* and *3*. The total number of such steps is *p=é(N-N')/ Dù-1*. To summarize, in each remaining step the following operations take place: (a) *D* tokens from the preceding execution leave the ERU, consuming *1* cycle; (b) *D* instructions are executed utilizing *(2-q)D* tokens, and consume *1* (or *D/E*) cycle; and (c) at the same time, *D* new instructions enter the ERU and consume *1* cycle. The instruction execution overlaps the transfer of new instructions, and therefore two cycles are consumed in step *i*, where *i=3, 4, 5, ..., p+2*. In the final step, *D* tokens leave the ERU consuming a single cycle. Thus, the total turnaround time is

$$T^{D2} = T_1^{D2} + T_2^{D2} + 2p + 1 = T_1^{D2} + T_2^{D2} + 2 \left| - \frac{N - N'}{\Delta} - \right| - 1.$$

4.3.3. Comparative Analysis of Turnaround Time

We assume, as earlier, that *N'=E* and *B=c* for the values of the parameters, and that the capacity *c* is identical for both designs. We also fit in the space of *c* words a number of instructions equal to the value of *E*, for smooth program execution on the D^2-CPU. We then obtain *c=(3-q)E*. Figure 3 shows results for the speedup achieved by the D^2-CPU when compared to the PC-driven design. The speedup is always greater than two. Actually, the performance improvement could be larger than that shown in Figure 3 because: (a) the capacity *c* in the D^2-CPU should be larger than the capacity in the PC-driven CPU because the former design has lower hardware complexity; and (b) the D^2-CPU can always take much better advantage of higher degrees of parallelism in application algorithms.

5. CONCLUSIONS

A new CPU design was proposed based on the observation that current PC-driven architectures do not match well with the requirements of applications. In their effort to improve performance and hide the aforementioned mismatch, PC-driven CPUs apply significant amounts of redundant operations, introduce hardware resources of low productive utilization, and increase power consumption. In contrast, our data-driven processor design matches well with the structure of application algorithms, yields outstanding performance, minimizes the number of redundant operations, and has lower power consumption.

REFERENCES

[1] J.A. Jacob and P. Chow, "Memory Interfacing and Instruction Specification for Reconfigurable Processors," *ACM Conf. FPGAs*, 1999, pp. 145-154.

[2] S.G. Ziavras, ``RH: A Versatile Family of Reduced Hypercube Interconnection Networks,'' *IEEE Trans. Paral. Distr. Syst.*, Vol. 5, No. 11, Nov. 1994, pp. 1210-1220.

[3] J.B. Dennis and D.P. Misunas, "A Preliminary Architecture for a Basic Data-Flow Processor," *Intern. Symp. Comput. Arch.*, 1975, pp. 125-131.

[4] G.M. Papadopoulos et al., "Monsoon: An Explicit Token-Store Architecture," *Int. Symp. Comp. Arch.*, 1990, pp. 82-91.

[5] W.-M. W. Hwu and Y.N. Patt, "HPSm, a High Performance Restricted Data Flow Architecture Having Minimal Functionality," *Intern. Symp. Comput. Arch.*, 1986.

[6] S.G. Ziavras et al., ``A Low-Complexity Parallel System for Gracious, Scalable Performance. Case Study for Near PetaFLOPS Computing," *6th Symp. Front. Mass. Paral. Comput.*, 1996, pp. 363-370.

[7] T. Golota and S.G. Ziavras, "A Universal, Dynamically Adaptable and Programmable Network Router for Parallel Computers," *VLSI Design*, Jan. 2001 (to appear).

[8] M. Gokhale, B. Holmes, and K. Iobst, "Processing In Memory: the Terasys Massively Parallel PIM Array," *Computer*, April 1995, pp. 23-31.

[9] M.S. Schlansker and B.R. Rau, "EPIC: Explicitly Parallel Instruction Computing," *Computer*, Febr. 2000, pp. 37-45.

[10] X. Tang and G.R. Gao, "Automatically Partitioning Threads for Multithreaded Architectures," *Journ. Paral. Distr. Comput.*, Vol. 58, 1999, pp. 159-189.

[11] G. Alverson, S. Kahan, and R. Korry, "Processor Management in the Tera MTA System," *Techn. Rep., Tera Comput.*, Seattle, WA, 1995.

[12] R. Korry et al., "Memory Management in the Tera MTA System," *Techn. Rep., Tera Comput.*, Seattle, WA, 1995.

[13] J. Hennessy, "The Future of Systems Research," *Computer*, Aug. 1999, pp. 27-33.

[14] H.H.J. Hum et al., "A Design Study of the Earth Multiprocessor," *Int'l. Conf. Par. Arch. Comp. Tech.*, 1995.

[15] D. Burger, J. Goodman, and A. Kagi, "Memory Bandwidth Limitations of Future Microprocessors," *23rd Int. Symp. Comput. Arch.*, May 1996.

[16] J.A. Fisher, "Very Long Instruction Word Architectures and the ELI-512," *10th Sym. Comp. Arch.*, 1983, pp. 140-150.

[17] Arvind and R.S. Nikhil, "Executing a Program on the MIT Tagged-Token Dataflow Architecture," *IEEE Trans. Comput.*, Vol. 39, No. 3, Mar. 1990, pp. 300-318.

[18] H. Terada et al., "Design Philosophy of a Data-Driven Processor: Q-p," *Journ. Inform. Proc.*, Vol. 10, No. 4, Mar. 1988, pp. 245-251.

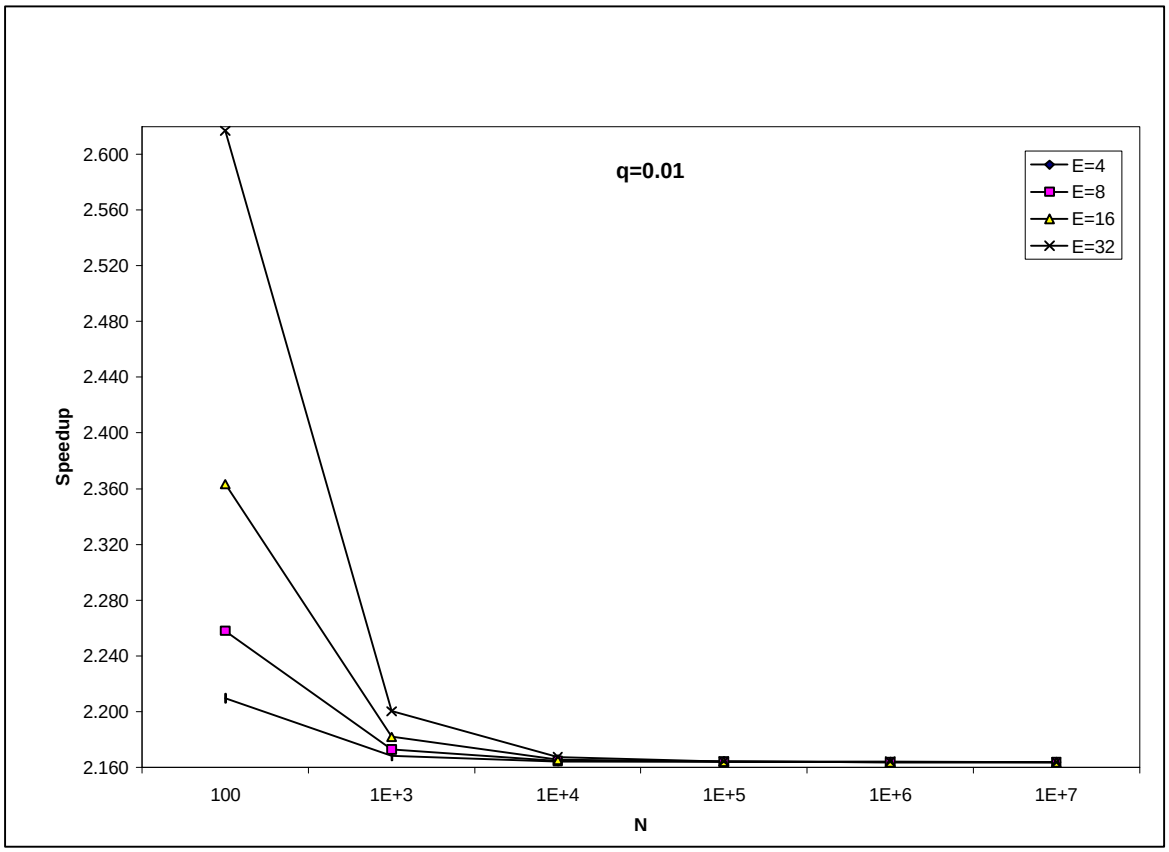

Figure 3. The speedup for *q=0.01*.